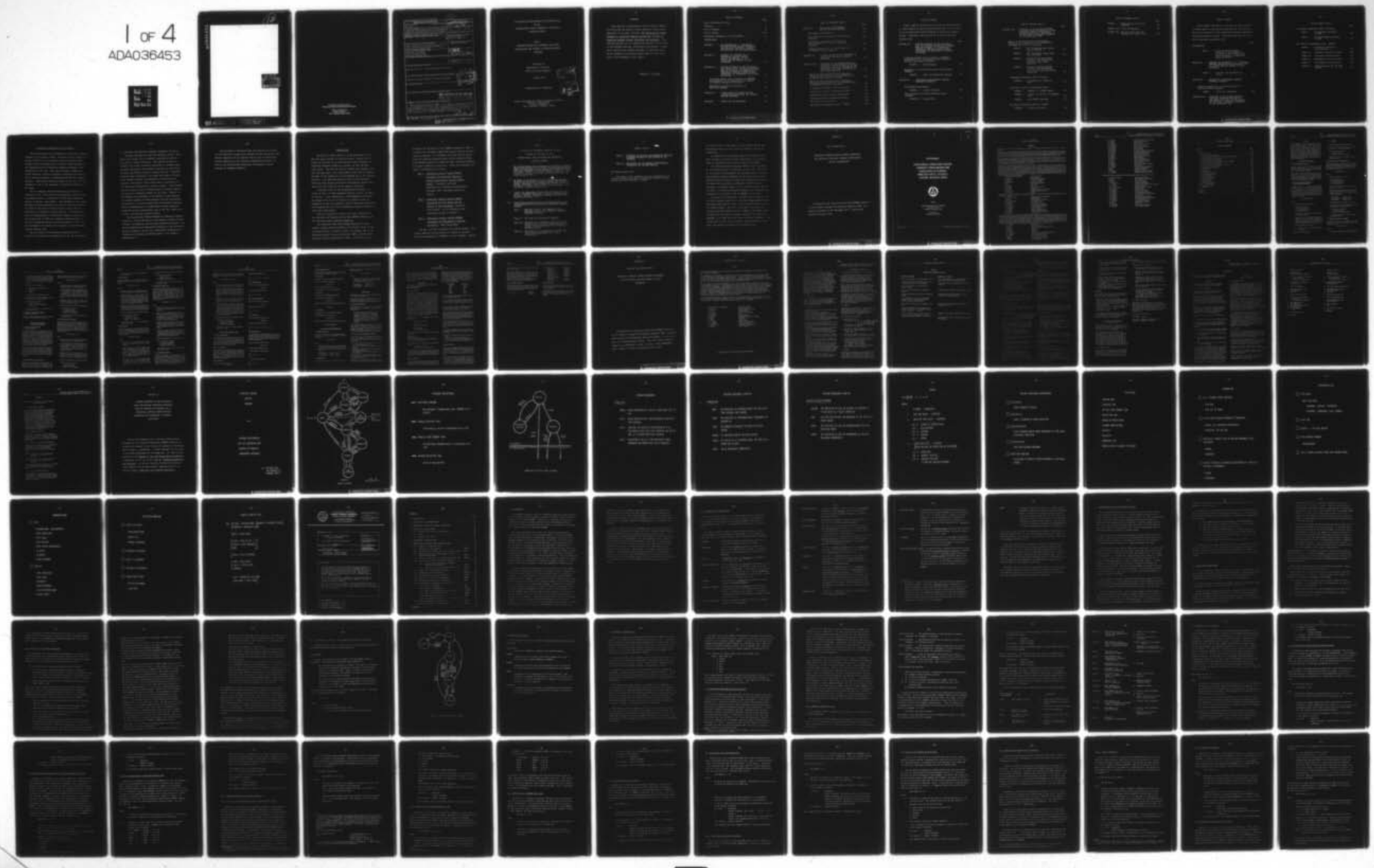
AD-A036 453 PURDUE UNIV LAFAYETTE IND PURDUE LAB FOR APPLIED IND--ETC F/G 9/2
SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIO--ETC(U)
JAN 77 N00014-76-C-0732

UNCLASSIFIED NL

1 OF 4
ADA036453



Code 23
a.s

(12)

ADA036453

DDC
RECEIVED
MAR 2 1977

BEST AVAILABLE COPY

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

This Report is Published as Part of
Engineering Experiment Station Bulletin 143 Series

Schools of Engineering
Purdue University
West Lafayette, Indiana 47907

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(12)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NR049-388 (6)

2. GOVT ACCESSION NO.:

3. RECIPIENT'S CATALOG NUMBER: (9) Final rept. 1 Feb 76- 31 Jan 77.

4. TITLE (and Subtitle): SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIONAL PURDUE WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS. PART I. EXTENDED FORTRAN FOR INDUSTRIAL REAL-TIME APPLICATIONS AND STUDIES ON PROBLEM-ORIENTED LANGUAGES.

5. TYPE OF REPORT & PERIOD COVERED: Final 2/1/76 - 1/31/77

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s):

8. CONTRACT OR GRANT NUMBER(s): (15) N00014-76-C-0732

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Purdue Laboratory for Applied Industrial Control
Schools of Engineering, Purdue University
West Lafayette, Indiana 47907

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
Department of the Navy
Office of Naval Research
Arlington, Virginia 22217

12. REPORT DATE: (11) Jan[illegible]77

13. NUMBER OF PAGES: 300 + xviii (12) 313p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report):

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report):

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

Approved for public release; distribution unlimited.

DDC
RECEIVED
MAR 7 1977
C

18. SUPPLEMENTARY NOTES:

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):

COPY AVAILABLE TO DDC DOES NOT PERMIT FULLY LEGIBLE PRODUCTION

20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

This volume represents Part 1 of a six volume set reproducing the major work accomplished by the International Purdue Workshop on Industrial Computer Systems during the past eight years. This material is reprinted from the Minutes of the several individual meetings of the Workshop and represents the work carried out by the standing committes of the Workshop.

408 244

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

BEST AVAILABLE COPY

SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIONAL PURDUE WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS

PART I

EXTENDED FORTRAN FOR INDUSTRIAL REAL-TIME APPLICATIONS AND STUDIES ON PROBLEM-ORIENTED LANGUAGES

Prepared for
Department of the Navy
Office of Naval Research

January 1977

Distribution is Unlimited

ACCESSION for
NTIS White Section ☑
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
DIST. AVAIL. and/or SPECIAL
A 23

(05)

Purdue Laboratory for Applied Industrial Control
Schools of Engineering
Purdue University
West Lafayette, Indiana 47907

FOREWORD

This material is published as part of Contract N00014-76-C-0732 with the Office of Naval Research, United States Department of the Navy, entitled, *The International Purdue Workshop on Industrial Computer Systems and Its Work in Promoting Computer Control Guidelines and Standards*. This contract provides for an indexing and editing of the results of the Workshop Meetings, particularly the Minutes, to make their contents more readily available to potential users. We are grateful to the United States Navy for their great help to this Workshop in this regard.

Theodore J. Williams

TABLE OF CONTENTS

Page

Report Documentation Page i

Foreword. v

List of Figures ix

List of Tables. xiii

Background Information on the Workshop. xv

INTRODUCTION. 1

SECTION I - ISA STANDARD S61.1 - INDUSTRIAL COMPUTER SYSTEM FORTRAN PROCEDURES FOR EXECUTIVE FUNCTIONS, PROCESS INPUT/OUTPUT, AND BIT MANIPULATION. . . 7

SECTION II - PROPOSED ISA STANDARD S61.2 - INDUSTRIAL COMPUTER SYSTEM FORTRAN PROCEDURES FOR FILE ACCESS AND THE CONTROL OF FILE CONTENTION. 21

SECTION III - WORKING DOCUMENTS OF THE INDUSTRIAL REAL-TIME FORTRAN COMMITTEE CONCERNING WORK ON PROPOSED ISA STANDARD S61.3, INDUSTRIAL COMPUTER SYSTEM FORTRAN PROCEDURES FOR MANAGEMENT OF PARALLEL ACTIVITIES. 31

A Process Control View of Tasking or FORTRAN Requirements for the Sequencing and Control of Parallel Concurrent Processes 33

Management of Parallel Activities in Real-Time FORTRAN. 49

SECTION IV - A REAL-TIME FILE SYSTEM AND JOB INITIALIZATION ROUTINES FOR PROCESS CONTROL COMPUTERS 95

SECTION V - FORMAT DEFINED LANGUAGES. 141

PRECEDING PAGE BLANK-NOT FILMED

TABLE OF CONTENTS (Cont.)

Page

SECTION VI - PRELIMINARY REQUIREMENTS PROBLEM-ORIENTED LANGUAGES 149

Preliminary Requirements. 151

User Criteria for Process-Oriented Source Language. 165

Preliminary Recommended Specification and Procedure for a Process Oriented Language. 183

Working Document for the Specification of a Problem Oriented Language 197

SECTION VII - JAPANESE REVIEW OF THE PROPOSALS OF THE PROBLEM-ORIENTED LANGUAGES COMMITTEE. 241

SECTION VIII - DECISIONS OF THE PROBLEM-ORIENTED LANGUAGES COMMITTEE RELATING TO THE TRANSLATOR METHOD OF PROCEEDING AND PERTINENT LITERATURE REFERENCES TO THE TECHNIQUES USES. 251

Report of the Problem Oriented Languages Committee, Sixth Workshop on Standardization of Industrial Computer Languages. 253

Report of the Problem Oriented Languages Committee, Seventh Workshop on Standardization of Industrial Computer Languages. 263

A Language Independent Macro Processor. 267

A Base for a Mobile Programming System. 275

Syntax Macros and Extended Translation. 279

Building a Mobile Programming System. 283

The ML/I Macro Processor. 287

The Mobile Programming System: STAGE2. 293

LIST OF FIGURES

Figure numbers used here are the same as those assigned to these figures in their previous publication in the Minutes of the International Purdue Workshop on Industrial Computer Systems. Therefore they will not be in strict numerical sequence here.

Page

SECTION III - WORKING DOCUMENTS OF THE INDUSTRIAL REAL-TIME FORTRAN COMMITTEE CONCERNING WORK ON PROPOSED ISA STANDARD S61.3, INDUSTRIAL COMPUTER SYSTEM FORTRAN PROCEDURES FOR MANAGEMENT OF PARALLEL ACTIVITIES

A Process Control View of Tasking or FORTRAN Requirements for the Sequencing and Control of Parallel Concurrent Processes

FIGURE 1 - *State Diagram* 35

Management of Parallel Activities In Real-Time FORTRAN

FIGURE 1 - State and Transition Diagram. . 64

SECTION VI - PRELIMINARY REQUIREMENTS PROBLEM-ORIENTED LANGUAGES

Preliminary Requirements

FIGURE 1 - Control Elements. 157

User Criteria for Process-Oriented Source Language

FIGURE V-1 - System View 167

LIST OF FIGURES (Cont.)

Page

SECTION VIII - DECISIONS OF THE PROBLEM-ORIENTED LANGUAGES COMMITTEE RELATING TO THE TRANSLATOR METHOD OF PROCEEDING AND PERTINENT LITERATURE REFERENCES TO THE TECHNIQUES USED

Report of the Problem Oriented Languages Committee Sixth Workshop on Standardization of Industrial Computer Languages

FIGURE 1 - POL Processing, Many Users - Single Computer 255

FIGURE 2 - POL Processing, Single User - Many Computers. 256

FIGURE 3 - Problem Oriented Language Converter and Translator - Statement Type Polcat Expansion 257

FIGURE 4 - Problem Oriented Language Convertor and Translator - Fill-in-the-Form Type Polcat Expansion 258

A Language Independent Macro Processor

FIGURE 1 - An Example of a Template Tree. 269

A Base for a Mobile Programming System

FIGURE 1 - Examples of SIMCMP Macros . . . 277

FIGURE 2 - Example of the Use of Parameter Zero. 277

FIGURE 3 - The SIMCMP Algorithm. 278

The Mobile Programming System: STAGE2

FIGURE 1 - Template Matching 294

LIST OF FIGURES (Cont.)

Page

FIGURE 2 - Organization of the Flub Computer. 296

FIGURE 3(a)- A Set of Templates. 298

FIGURE 3(b)- The Tree Built From the Templates of Figure 3(a). . . . 298

LIST OF TABLES

Table numbers used here are the same as those assigned to these tables in the previous publications of these documents in the Minutes of the International Purdue Workshop on Industrial Computer Systems. Therefore they will not be in strict numerical succession here.

Page

INTRODUCTION

TABLE I - A List of All Documents Produced in This Summary of the Work of the International Purdue Workshop on Industrial Computer Systems 3

SECTION II - PROPOSED ISA STANDARD S61.2 - INDUSTRIAL COMPUTER SYSTEM FORTRAN PROCEDURES FOR FILE ACCESS AND THE CONTROL OF FILE CONTENTION

TABLE I - Features and Attributes of Files. 25

SECTION VI - PRELIMINARY REQUIREMENTS PROBLEM-ORIENTED LANGUAGES

Working Document for the Specification of a Problem Oriented Language

TABLE 2 - Value of a Subscript 239

SECTION VIII - DECISIONS OF THE PROBLEM-ORIENTED LANGUAGES COMMITTEE RELATING TO THE TRANSLATOR METHOD OF PROCEEDING AND PERTINENT LITERATURE REFERENCES TO THE TECHNIQUES USES. . . . .

PRECEDING PAGE BLANK-NOT FILMED

LIST OF TABLES (Cont.)

Page

A Language Independent Macro Processor

TABLE I - String Names Available in LIMP. 269

TABLE II - Pattern Elements Accepted by LIMP. 270

The Mobile Programming System: STAGE2

TABLE I - Characters Defined by the Flag Line. 294

TABLE II - Parameter Conversion Digits. . . 295

TABLE III - Processor Function Digits. . . . 295

TABLE IV - Breakdown of the Flub Word . . . 296

TABLE V - Flub Machine Instructions. . . . 296

TABLE VI - Implementation for CDC 6000 Series 297

BACKGROUND INFORMATION ON THE WORKSHOP

The International Purdue Workshop on Industrial Computer Systems, in its present format, came about as the result of a merger in 1973 of the Instrument Society of America (ISA) Computer Control Workshop with the former Purdue Workshop on the Standardization of Industrial Computer Languages, also cosponsored by the ISA. This merger brought together the former workshops' separate emphases on hardware and software into a stronger emphasis on engineering methods for computer projects. Applications interest remains in the use of digital computers to aid in the operation of industrial processes of all types.

The ISA Computer Control Workshop had itself been a renaming in 1967 of the former Users Workshop on Direct Digital Computer Control, established in 1963 under Instrument Society of America sponsorship. This Workshop in its annual meetings had been responsible for much of the early coordination work in the field of direct digital control and its application to industrial process control. The Purdue Workshop on Standardization of Industrial Computer Languages had been established in 1969 on a semiannual meeting basis to satisfy a widespread desire and need expressed at that time for development of standards for languages in the industrial computer control area.

The new combined international workshop provides a forum for the exchange of experiences and for the development of guidelines and proposed standards throughout the world.

Regional meetings are held each spring in Europe, North America and Japan, with a combined international meeting each fall at Purdue University. The regional groups are divided into several technical committees to assemble implementation guidelines and standards proposals on specialized hardware and software topics of common interest. Attendees represent many industries, both users and vendors of industrial computer systems and components, universities and research institutions, with a wide range of experience in the industrial application of digital systems. Each workshop meeting features tutorial presentations on systems engineering topics by recognized leaders in the field. Results of the workshop are published in the Minutes of each meeting, in technical papers and trade magazine articles by workshop participants, or as more formal books and proposed standards. Formal standardization is accomplished through recognized standards-issuing organizations such as the ISA, trade associations, and national standards bodies.

The International Purdue Workshop on Industrial Computer Systems is jointly sponsored by the Automatic Control Systems Division, the Chemical and Petroleum Industries Division, and the Data Handling and Computations Division of the Instrument Society of America, and by the International Federation for Information Processing as Working Group 5.4 of Technical Committee TC-5.

The Workshop is affiliated with the Institute of Electrical and Electronic Engineering through the Data Acquisition and Control Committee of the Computer Society and the Industrial Control Committee of the Industrial Applications Society, as well as the International Federation of Automatic Control through its Computer Committee.

INTRODUCTION

The Office of Naval Research of the Department of the Navy has made possible an extensive report, summary and indexing of the work of the International Purdue Workshop on Industrial Computer Systems as carried out over the past eight years. The work has involved twenty-five separate workshop meetings plus a very large number (over 100) of separate meetings of the committees of the workshop and of its regional branches. This work has produced a mass of documentation which has been severely edited for the original minutes themselves and then again for these summary collections.

A listing of all of the documentation developed as a result of the U. S. Navy sponsored project is given in Table I at the end of this Introduction. The workshop participants are hopeful that it will be helpful to others as well as themselves in the very important work of developing guidelines and standards for the field of industrial computer systems in their many applications.

This part reports the work of two major committees of the Workshop, the Industrial Real-Time FORTRAN Committee and the Problem-Oriented Languages Committee.

The Industrial Real-Time FORTRAN Committee has been working closely with the Standards and Practice Board of the Instrument Society of America (ISA), and through them to the American National Standards Institute (ANSI), and the International Standards Organization (ISO), to develop a set of standards for extensions to the FORTRAN language to make it a capable and worthwhile industrial process control and data handling language. To facilitate that work ISA has constituted the American Regional Branch of the Industrial Real-Time FORTRAN Committee as Committee SP61, Industrial FORTRAN, to develop standards in this area. Three such standards are issed or contemplated as follows:

S61.1 - Industrial Computer System FORTRAN Procedures for Executive Functions, Process Input/Output, and Bit Manipulation. Originally published September 1972, Revised and Republished September 1976, Instrument Society of America.

S61.2 - Industrial Computer System FORTRAN Procedures for File Access and the Control of File Contention. Presently under Review. To be Published by the Instrument Society of America.

S61.3 - Industrial Computer System FORTRAN Procedures for Management of Parallel Activities. Under Development.

The S61.1 and S61.2 documents are reprinted here. The latest committee working papers on tasking or parallel activities management in FORTRAN are also included. Because

TABLE I

A LIST OF ALL DOCUMENTS PRODUCED IN THIS SUMMARY OF THE WORK OF THE INTERNATIONAL PURDUE WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS

1. The International Purdue Workshop on Industrial Computer System and Its Work in Promoting Computer Control Guidelines and Standards, Report Number 77, Purdue Laboratory for Applied Industrial Control, Purdue University, West Lafayette, Indiana, Originally Published May 1976, Revised November 1976.

2. An Index to the Minutes of the International Purdue Workshop on Industrial Computer Systems and Its Predecessor Workshops, Report Number 88, Purdue Laboratory for Applied Industrial Control, Purdue University, West Lafayette, Indiana, January 1977.

3. A Language Comparison Developed by the Long Term Procedural Languages Committee - Europe, Committee TC-3 of Purdue Europe, Originally Published January 1976, Republished October 1976.

4-9. Significant Accomplishments and Documentation of the International Purdue Workshop on Industrual Computer Systems.

Part I - Extended FORTRAN for Industrial Real-Time Applications and Studies in Problem Oriented Languages.

Part II - The Long Term Procedural Language.

Part III- Developments in Interfaces and Data Transmission, in Man-Machine Communications and in the Safety and Security of Industrial Computer Systems.

Part IV - Some Reports on the State of the Art and Functional Requirements for Future Applications.

TABLE I (Cont.)

Part V - Documents on Existing and Presently Proposed Languages Related to the Studies of the Workshop.

Part VI - Guidelines for the Design of Man/Machine Interfaces for Process Control

All dated January 1977.

The latter seven documents are also published by the Purdue Laboratory for Applied Industrial Control, Purdue University, West Lafayette, Indiana.

of the existence of the final or near final documents the development papers on the first two standards are not included here.

An excellent file handling system for industrial computer systems developed by Royal Dutch Shell in the Netherlands is reproduced here from the 1970 Fall Workshop.

The Problem-Oriented Language Committee has its choice of two possible modes of operation in this area. The first is to propose the development and standardization of one or more problem-oriented languages in any one or more of the several possible formats in which such languages can appear. The second is to develop a mechanism by which any such language, in whatever format, can be translated into one of the major high level procedural languages now undergoing the standardization process, such as FORTRAN or the LTPL.

Because of the extremely difficult task of carrying out the first proposal above due to the very large number of the languages which already exist, it has been decided to pursue the second avenue. The key to such work is the use of a string processing language such as STAGE 2, described herein, to translate the subject language, whatever its form, into FORTRAN or the LTPL once this latter has been developed. The second part of this volume contains several committee documents and reports concerning these possibilities.

SECTION I

ISA Standard S61.1

INDUSTRIAL COMPUTER SYSTEM FORTRAN PROCEDURES FOR EXECUTIVE FUNCTIONS, PROCESS INPUT/OUTPUT, AND BIT MANIPULATION

Developed by the Industrial Real-Time FORTRAN Committee of the Workshop through ISA Standards Committee SP61. Originally published by ISA September 1972. Revised and reissued September 1976.

PRECEDING PAGE BLANK-NOT FILMED

STANDARD

INDUSTRIAL COMPUTER SYSTEM FORTRAN PROCEDURES FOR EXECUTIVE FUNCTIONS, PROCESS INPUT/OUTPUT, AND BIT MANIPULATION

Sponsor

INSTRUMENT SOCIETY OF AMERICA
400 Stanwix Street
Pittsburgh, Pennsylvania 15222

Issued: 1972
Revised: February 1976

Printed in U.S.A.

Copyright 1976 by the Instrument Society of America

PRECEDING PAGE BLANK-NOT FILMED

PREFACE

This Preface, all footnotes and all Appendices are included for informational purposes and are not part of Standard ISA-S61.1.

This Standard has been prepared as a part of the service of the Instrument Society of America toward a goal of uniformity in the field of instrumentation. To be of real value this document should not be static but should be subjected to periodic review. Toward this end the Society welcomes all comments and criticisms and asks that they be addressed to the Standards and Practices Board Secretary, Instrument Society of America, 400 Stanwix Street, Pittsburgh, Pennsylvania 15222.

The ISA Standards Committee on Industrial Computer System FORTRAN Procedures SP61, operates within the ISA Standards and Practices Department, W. B. Miller, Vice President. The SP61 Committee also is a working group of the FORTRAN Committee of the International Purdue Workshop on Industrial Computer Systems which provides the impetus for the development of this and related standards. The FORTRAN Committee is chaired by Ms. Maxine N. Hands; General Chairman of the International Purdue Workshop is Dr. T. J. Williams.

Committee SP61, Industrial Computer System FORTRAN Procedures was established in June 1971 as a working group of the FORTRAN Committee of the Purdue Workshop on Standardization of Industrial Computer Languages (later reorganized and renamed the International Purdue Workshop on Industrial Computer Systems). As a result of their activity, ISA Standard S61.1-1972, Industrial Computer System FORTRAN Procedures for Executive Functions and Process Input Output, was published by the ISA in 1972. The committee continued to work on additional draft standards as part of the International Purdue Workshop. In 1974, extensive use of S61.1-1972 and discussions with ANSI X3J3 FORTRAN Committee indicated a need for revision. This revision was accomplished by a working group of the Purdue Workshop FORTRAN Committee which constituted the SP61 Committee listed below:

M. R. Gordon-Clark, Chairman	Scott Paper Company
A. Arthur	IBM Corporation
F. E. Bearden	Modular Computer Systems
R. H. Caro	The Foxboro Company
L. M. Cartright	Inland Steel Company
W. Diehl	Hewlett-Packard Company
D. Frost	Honeywell, Inc.
M. N. Hands	Datum, Inc.
T. L. Luekens	Johnson Service Company
J. McGovney	Interdata, Inc.
O. Petersen	Norwegian Institute of Technology
W. A. Resch, III	Tennessee Eastman Company
D. W. Zobrist	ELDEC

The members of the SP61 Committee who developed the original S61.1-1972 Standard and their affiliations at the time of its approval are as follows:

E. A. Kelly, Chairman	Kaiser Aluminum & Chemical Corp.
A. J. Arthur	IBM Corporation
W. Diehl	Digital Equipment Corporation
M. R. Gordon-Clark	Scott Paper Company
M. N. Hands	Honeywell, Inc.
R. E. Hohmeyer	Control Data Corporation
P. H. Jarvis	General Electric Corporation
L. J. Lockwood	General Motors Corporation
P. K. Mattheiss	Sun Oil Company
R. S. Moser	Applied Automation, Inc.
G. W. Oerter	Leeds & Northrup Company
R. Osmundsen	Technical University of Norway
S. Taylor	Honeywell, Inc.

In preparing the revised S61.1-1976, the committee received extensive assistance and guidance from the members of the International Purdue Workshop and the ANSI X3J3 FORTRAN Committee. Well over 100 people contributed to this revised standard through technical discussions, critiques of drafts, by offering suggestions towards its improvement, and in other ways. The contributions of all of these people are greatly acknowledged. In addition to the SP61 Committee and International Purdue Workshop members, the following people have formally reviewed this revised standard as a Board of Review. They have indicated their general concurrence with this standard; however, it should be noted that they have acted as individuals and their approval does not necessarily constitute approval by their company or facility.

J. J. Anderson	Watermation
J. C. Archer	Gilbert Associates, Inc.
J. Barnard	NY State Agricultural Experiment Station
D. Braedley	Polysar Ltd.
J. A. Conover	Monsanto
R. L. Curtis	ALCOA
D. A. DeMattia	Kaiser Aluminum and Chemical Corporation
D. G. Dimmler	Brookhaven National Laboratory
C. L. Edwards	El Paso Products Company
R. Egger	Boeing Aerospace Company
V. Elischer	Lawrence Berkeley Laboratory
G. F. Erk	Sun Oil Company

R. W. Gellie	National Research Council of Canada
J. C. Houser	Simpson Timber Company
T. J. Harrison	IBM Corporation
W. R. Hodson	Leeds and Northrup Company
R. E. Hohmeyer	Control Data Corporation
D. L. Hutchins	Procter and Gamble Company
T. S. Imsland	Fischer Controls
P. Kushkowski	Northeast Utilities
R. F. Laird, Jr.	E. I. duPont de Nemours & Company
F. R. Lenkszus	Argonne National Laboratory
A. C. Lumb	Procter and Gamble Company
R. G. Marvin	Dow Chemical Company
B. B. Misra	Babcock and Wilcox Company
J. H. Morrison	Fuller Engineering
N. Moseley, Jr.	Merck & Company, Inc.
A. W. Petty	Procter and Gamble Company
D. Rosich	City College of New York
E. H. Spencer	Exxon Corporation
R. J. Spitznas	Baltimore Gas & Electric Company
N. G. Sutter	Honeywell, Inc.
R. F. Thomas	Los Alamos Scientific Laboratory
J. Tsing	Bechtel Power Corporation
J. White	University of Arizona
R. G. Wilhelm	Industrial Nucleonics Corporation
F. G. Willard	Westinghouse Electric Corporation

This Revised Standard was approved by the ISA Standards and Practices Board on June 1, 1976.

W. B. Miller, Chairman	Moore Products Company
P. Bliss	Pratt & Whitney Aircraft Company
E. J. Byrne	Brown & Root, Inc.
W. Calder	The Foxboro Company
B. A. Christensen	Continental Oil Company
L. N. Combs	Retired E. I. duPont deNemours & Company
R. L. Galley	Bechtel Corporation
R. G. Hand, Secretary	Instrument Society of America
T. J. Harrison	IBM Corporation
H. T. Hubbard	Southern Services, Inc.
T. S. Imsland	Fisher Controls Company
P. S. Lederer	National Bureau of Standards
O. P. Lovett, Jr.	E. I. duPont deNemours & Company
E. C. Magison	Honeywell Inc.
R. L. Martin	Tex-A-Mation Engineering Inc.
A. P. McCauley	Glidden Durkee Div. SCM Corporation
T. A. Murphy	The Fluor Corporation, Ltd.
R. L. Nickens	Reynolds Metals Company
G. Platt	Bechtel Power Corporation
A. T. Upfold	Polysar Ltd.
K. A. Whitman	Allied Chemical Corporation
J. R. Williams	Stearns-Roger Corporation

TABLE OF CONTENTS

Section Page

1. Scope 13
2. Executive Interface 13
   2.1 Control of Program Execution 13
   2.2 Starting a Program Immediately or After a Specified Time Delay 13
   2.3 Starting a Program At a Specified Time 13
   2.4 Delaying Continuation of a Program 14
   2.5 Program Termination Accomplished Using Present Standard 14
3. Process Input/Output Function Interfaces 14
   3.1 Control of Analog and Digital Sensors and Outputs 14
   3.2 Analog Inputs in a Sequential Order 14
   3.3 Analog Inputs in Any Sequence 14
   3.4 Analog Output 15
   3.5 Digital Input 15
   3.6 Momentary Digital Output 15
   3.7 Latching Digital Output 16
4. Bit String Manipulation 16
   4.1 Types of Manipulation 16
   4.2 Logical Operations 16
   4.3 Shift Operations 17
   4.4 Bit Testing and Setting 17
5. Date and Time Information 17
   5.1 Obtain Time of Day 17
   5.2 Obtain Date 18

Appendix A 18
Appendix B 18

1 SCOPE

This standard presents external procedure references for use in industrial computer systems. These external procedure references permit interface of programs with executive routines, process input and output functions, allow manipulation of bit strings and provide access to time and date information. These procedures are intended for use with programs written in FORTRAN conforming to the International Standards Organization (ISO) Programming Language-FORTRAN R1539-1972.[1] expected to be executing both in a solitary and in a multiprogramming environment under the control of a real time executive routine.

2 EXECUTIVE INTERFACE

2.1 Control of Program Execution

Executive interfaces provide the facility to control operation of the programs within the system. Through these external procedures, one may start, stop or delay the execution of application programs.

The argument m, shown below, shall be set equal to or greater than two (2) in value for all instances in which the request is not accepted by the executive routine. Individual implementations may specify unique values of m within the allowable range to designate the specific reason for which the request was rejected.

2.2 Starting a Program Immediately or After a Specified Time Delay

Execution of a reference to the subroutine START shall, after the expiration of the specified time delay, cause the execution of the designated program. The actual time delay obtainable in a specific industrial computer system is subject to the resolution of that system's real time clock. Execution of the designated program will commence at the program's first executable statement. The form of this call is:

CALL START (i, j, k, m)

where:

i specifies the program to be executed.

The argument is either:
a) an integer expression
or b) an integer array name
or c) a procedure name

The processor shall define which of the above three forms is acceptable.

j specifies the minimum length of time, in units as specified by k, to delay before executing the program. If the value of j is zero or negative, the requested program will be run as soon as permissible. This argument shall be an integer expression.

k specifies the units of time as follows:

0 — Basic counts of the system's real time clock
1 — Milliseconds
2 — Seconds
3 — Minutes

This argument shall be an integer expression.

m is set on return to the calling program to indicate the disposition of the request as follows:

The value must be 1 or greater
1 — Request accepted
2 or greater — Request not accepted

This argument shall be an integer variable or integer array element.

2.3 Starting a Program at a Specified Time

Execution of a reference to the subroutine TRNON shall cause the designated program to be executed at a specified time of day. Execution of the designated program will commence at the program's first executable statement. The form of this call is

CALL TRNON (i, j, m)

where:

i specifies the program to be executed.

The argument is either:
a) an integer expression
or b) an integer array name
or c) a procedure name

The processor shall define which of the above three forms is acceptable.

j designates an array whose first three (3) elements contain the absolute time of day at which the specified program is to be executed. These elements are as follows:

First element — Hours (0 to 23)
Second element — Minutes (0 to 59)
Third element — Seconds (0 to 59)

This argument shall be an integer array name.

m is set on return to the calling program to indicate the disposition of the request as follows:

The value must be 1 or greater
1 — Request accepted
2 or greater — Requested rejected

This argument shall be an integer variable or integer array element.

2.4 Delaying Continuation of a Program

Execution of a reference to the subroutine WAIT shall return after a delay of a specified length of time. The form of this call is:

CALL WAIT (j, k, m)

where:

j specifies the length of time in units as specified by k to delay before returning to the calling procedure. If the value is zero or negative, no delay will occur. Limitations of the implementation shall not cause the precise time to be less than requested. This argument shall be an integer expression.

---

[1] ANSI X3.9 — 1966 FORTRAN

k specifies units of time as follows:

0 — Basic counts of the system's clock
1 — Milliseconds
2 — Seconds
3 — Minutes

This argument shall be an integer expression.

m is set on return to the calling program to indicate the disposition of the request as follows:

The value must be 1 or greater
1 — Request accepted
2 or greater — Delay as specified has not occurred.

This argument shall be an integer variable or integer array element.

2.5 Program Termination Accomplished Using Present Standard

Termination of programs shall be accomplished using the STOP statement from (ISO) R1539 — 1972.

3 PROCESS INPUT/OUTPUT FUNCTION INTERFACES

3.1 Control of Analog and Digital Sensors and Outputs

Process input-output function interfaces allow access to data related to specific analog and digital sensors and outputs. The execution of references to these subroutines will result in return only being made to the requesting program when the requested data transfer is complete, or when the requested operation is terminated. The argument m, shown below, is to be interrogated by the requesting program in order to determine the status of the request. An individual processor may specify unique values for m within the allowable range to designate the specific error conditions.

The results of the input operation and the data presented for output operations are processor dependent. The operations described are intended to be unformatted transfers to and from the specified storage in the processor. Therefore, the representation in the storage of the processor is an image of the input or output device specified.

3.2 Analog Inputs in a Sequential Order

Execution of a reference to this subroutine allows the input of data from any number of analog points in a sequence which is specified by the processors input hardware interface. The argument j specifies the first point to be read and also the particular form of all the data placed in the array k. The form of this call is:

CALL AISQW (i, j, k, m)

where:

i specifies the number of analog points to be read. This argument shall be an integer expression.

j specifies hardware or software acquisition and conversion information. Specific information relevant to construction of this argument is processor dependent because various types of analog-to-digital systems are available. This argument shall be either an integer expression or an integer array name.

k designates an array to which the values are assigned. This argument shall be an integer array name, or an integer array element.

m indicates the disposition of the request as follows:

The value must be 1 or greater.
1 — All data collected
2 — Operation incomplete
3 or greater — Error conditions

This argument shall be an integer variable or an integer array element.

3.3 Analog Inputs in any Sequence

Execution of references to this subroutine allows the input of data from analog points in a sequence which is independent of the hardware. This may also be referred to as the input of analog data in a random order. The array j controls the input sequence and allows the specification of hardware or software acquisition and conversion information on an individual point basis. The form of this call is:

CALL AIRDW (i, j, k, m)

where:

i specifies the number of analog points to be read. This argument shall be an integer expression.

j specifies hardware or software acquisition and conversion information for each analog input signal. Specific information relevant to construction of this argument is processor dependent. This argument shall be an integer array name, or an integer array element.

k designates an array to which the values are assigned. The order of the elements in k will correspond to the order in j. This argument shall be an integer array name, or an integer array element.

m indicates the disposition of the request as follows:

The value must be 1 or greater
1 — All data collected
2 — Operation incomplete
3 or greater — Error conditions

This argument shall be an integer variable or integer array element.

3.4 Analog Output

Execution of references to this subroutine allows the output of analog signals in any sequence. The form of this call is:

CALL AOW (i, j, k, m)

where:

i specifies the number of analog points to be written. This argument shall be an integer expression.

j specifies hardware or software conversion and transmission information for the analog output point. Specific information relevant to construction of this argument is processor dependent because various types of digital-to-analog systems are available. This argument shall be an integer array name, or an integer array element.

k designates an array from which the analog output values are taken. The order of the elements in k will correspond to the order j. This argument shall be an integer array name or integer array element.

m indicates the disposition of the request as follows:

The values must be 1 or greater
1 — All data outputted
2 — Operation incomplete
3 or greater — Error conditions

This argument shall be an integer variable or integer array element.

3.5 Digital Input

Execution of references to this subroutine causes the input of process information coded as a set of bits. A set of bits is typically organized and read as an external digital word whose length is processor dependent. The form of this call is:

CALL DIW (i, j, k, m)

where:

i specifies the number of external digital words to be read. This argument shall be an integer expression.

j specifies hardware or software acquisition and conversion information for each digital input word. Specific information relevant to construction of this argument is processor dependent because various types of digital inputs are available. This argument shall be an integer array name, or integer array element.

k designates an array to which the requested values are assigned. The order of the elements in k will correspond to the order in j. This argument shall be an integer array name, or integer array element name.

m indicates the disposition of the request as follows:

The value must be 1 or greater.
1 — All data collected
2 — Operation incomplete
3 or greater — Error conditions

This argument shall be an integer variable or integer array element.

3.6 Momentary Digital Output

The execution of references to this subroutine causes the output of momentary digital signals. These signals consist of sets of bits typically organized as an external digital word whose length is processor dependent. This type of output is characterized by an action which momentarily sets individual outputs when a corresponding bit in the output data is set. These output bits will then be reset after a specific time period. The form of this call is:

CALL DOMW (i, j, k, n, m)

where:

i specifies the number of external digital words to be written. This argument shall be an integer expression.

j specifies hardware transmission information for each word of output. Specific information relevant to construction of this argument is processor dependent because various types of digital outputs are available. This argument shall be an integer array name, or an integer array element.

k designates an array whose contents are the image words to be written. The order of the elements in k will correspond to the order in j. This argument shall be an integer array name, or an integer array element.

n specifies the duration, measured in basic system clock counts, that the outputs are to remain set. If the processor does not allow selection of duration, this argument is ignored but must be present. This argument shall be an integer expression.

m indicates the disposition of the request as follows:

The value must be 1 or greater
1 — All outputs accomplished
2 — Operation incomplete
3 or greater — Error conditions

This argument shall be an integer variable or integer array element.

3.7 Latching Digital Output

Execution of references to this subroutine causes the output of digital signals which can be latched in either the set or reset state. These signals consist of sets or reset state. These signals consist of sets of bits typically organized as an external digital word whose length is processor dependent. This type of output is characterized by an action which sets individual outputs when a corresponding bit in the output is set and clears individual outputs when a corresponding bit is reset. The form of this call is:

CALL DOLW (i, j, k_1, k_2, m)

where:

i	specifies the number of digital words to be written. This argument shall be an integer expression.
j	specifies hardware transmission information for each word of digital output. Specific information relevant to construction of this argument is processor dependent. This argument shall be an integer array name, or an integer array element name.
k_1	designates an array whose values are the image words to be output. This argument shall be an integer array name, or an integer array element.
k_2	designates an array whose values define digital outputs which can be changed by the subroutine. A bit set in the k_2 array indicates that the digital output will be changed to the state defined by the corresponding bit position in the corresponding integer array element in k_1. The order of the elements in k_1 and k_2 will correspond to the order in j. This argument shall be an integer array name, or an integer array element.
m	indicates the disposition of the request as follows: The value must be 1 or greater 1 — All outputs accomplished 2 — Operation incomplete 3 or greater — Error conditions This argument shall be an integer variable or integer array element.

4 BIT STRING MANIPULATION

4.1 Types of Manipulation

The subprograms which follow allow the programmer to view integer data as ordered sets of bits (a_n, a_n-1, ... a_0), where the set is a place positional binary representation of an integer value, thus permitting interrogation and manipulation of integers on a bit-by-bit basis. The value of n is processor defined.

4.2 Logical Operations

These operations are external functions. In the following functions, j and m are integer expressions. Operations are performed on all bits which represent the value of an integer internal to the processor. Operations are done bit-by-bit on corresponding bits, that is, the corresponding bits of the actual arguments j and m are used to generate the integer result.

4.2.1 Inclusive Or

The form of this function reference is:

IOR (j, m)

where the result of IOR (j, m) is:

$$\sum_{k=0}^{n} 2^k * (j_k + m_k - (j_k * m_k))$$

4.2.2 Logical Product

The form of this function reference is:

IAND (j, m)

where the result of IAND (j, m) is:

$$\sum_{k=0}^{n} 2^k * (j_k * m_k)$$

4.2.3 Logical Complement

The form of this function reference is:

NOT (j)

where the result of NOT (j) is:

$$\sum_{k=0}^{n} 2_k * (1-j_k)$$

4.2.4 Exclusive Or

The form of this function reference is:

IEOR (j, m)

where the result of IEOR (j, m) is:

$$\sum_{k=0}^{n} 2^k * (2 - (j_k + m_k)) * (j_k + m_k)$$

4.3 Shift Operations

This operation is an external function. In the following function j and m are integer expressions. Operations are performed on all bits which represent the value of an integer internal to the processor, and are used to generate an integer result.

The form of this function reference is:

ISHFT (j, m)

where the result of ISHFT (j, m) is:

If the value of m is positive or zero

$$\sum_{k=0}^{n-m} 2^{k+m} * j_k$$

If the value of m is negative

$$\sum_{k=m}^{n} 2^{k+m} * j_k$$

4.4 Bit Testing and Setting

These operations are external functions. In the following functions j and m are integer expressions.

4.4.1 Bit Test

This logical function tests a specified bit of an integer.

The form of this function reference is:

BTEST (j, m)

where the result of BTEST (j, m) is:

if IAND (j,2^m) = 0 then FALSE, else TRUE

4.4.2 Bit Set

This function sets a specified bit of an integer.

The form is this function reference is:

IBSET (j, m)

where the result of the function reference IBSET (j, m) is:

IOR (j, 2^m)

4.4.3 Bit Clear

This function clears a specified bit of an integer.

The form of this function reference is:

IBCLR (j, m)

where the result of the function reference IBCLR (j, m) is:

IAND (j, NOT(2^m))

5 DATE AND TIME INFORMATION

5.1 Obtain Time of Day

Execution of references to this subroutine allows a program to determine the current time of day. The form of this call is:

CALL TIME (j)

where:

j designates an integer array into whose first three (3) elements the absolute time of day will be placed. The contents of these elements shall be as follows:

First Element — Hours 0 to 23
Second Element — Minutes 0 to 59
Third Element — Seconds 0 to 59

5.2 Obtain Date

Execution of references to this subroutine allows a program to determine the current calendar date. The form of this call is:

CALL DATE (j)

where:

j designates an integer array into whose first three (3) elements the date will be placed. The contents of these elements shall be as follows:

First Element — AD year since zero
Second Element — Month 1 to 12
Third Element — Day 1 to 31

APPENDIX A

(This appendix is not part of the ISA Standard S61.1, but is included to facilitate its use.)

Considerations leading to "ISA-S61.1 Industrial Computer System FORTRAN Procedures for Executive Functions Process Input/Output, and Bit-Manipulation."

A.1 Historical Development

This standard is a direct outgrowth of the International Purdue Workshop on Industrial Computer Systems whose goals are:

> *To make the definition, justification,* hardware and software design, procurement, programming, installation, commissioning, operation and maintenance of industrial computer systems more efficient and economical through the development of standards and/or guidelines on an international basis.

The Workshop formed several committees to achieve their objectives. The FORTRAN Committee was charged with the task of preparing a set of Industrial Process Standards compatible with FORTRAN as defined by ISO-R1539-1972. This standard is a result of that committee's work. Additional standards are being developed.

A.2 Criteria Used in Developing This Standard

The committee assessed that FORTRAN was the most widely used language in the industrial environment. It thus used the following guidelines in the development of the standard.

1. The standard should cover features commonly used by existing industrial computer systems.
2. The standards should be easy to implement for most vendors.
3. The standards should follow the syntax and intent of FORTRAN as defined by ISO.
4. The standards should not restrict the future evolution of FORTRAN.

The development of the FORTRAN language is presently the responsibility of ISO who has delegated this responsibility to ANSI X3J3.

In order that these standards comply with the ISO-R1539-1972 standard as far as possible, external procedure references were used rather than direct changes or additions to the syntax of FORTRAN. This does not imply that this is the only way to provide these features nor does it exclude the possibility or desirability that the language will be developed through syntax changes so that these features and other related features can be performed.

APPENDIX B

(This appendix is not part of the ISA Standard S61.1, but is included to facilitate its use.)

NOTES BY SECTION

B.1 Section 2 Notes

This standard is a permissive standard in that it does not prescribe how the executive will respond to external procedure references nor does it describe how the information is passed to the executive routine. In particular, the argument "i" in CALL START (2.2) and CALL TRNON (2.3) has three forms, an integer expression, an integer array name or a program name, only one of which is permissible in any particular program. This restriction is a necessary consequence of the requirements of the FORTRAN language that any argument which is an external procedure reference be of a defined type; integer expressions, integer array names and program names are different types (Section 8.4.2 of ISO-R1539-1972). It is also a consequence of the FORTRAN language that if the argument is a program name, this name must appear in an EXTERNAL statement (Section 7.2.15 of ISO-R1539-1972).

Examples of the use of the argument i as an integer expression are:

```
        CALL START (7,0,0,M)

        DATA J/7/
        CALL START (J,0,0,M)

        DATA J/2HAB/
        CALL START (J,0,0,M)
```

An example of the use of the argument i as an integer array name is:

```
INTEGER XYZ
DIMENSION  XYZ (3)
DATA XYZ(1), XYZ(2),XYZ(3)/2HAB,2HCD,2HEF/
CALL    START    (XYZ,0,0,M)
```

An example of the use of the argument i as a procedure name is:

```
        EXTERNAL    ABCD
        CALL     START    (ABCD,0,0,M)
```

Interchangeability of programs between different processors is reduced by permitting three possible types for this argument but the committee feels it is premature to standardize to achieve full interchangeability in this area.

B.2 Section 3 Notes

As an extension to the process I/O calls, six additional calls may be defined. They are AISQ, AIRD, AO, DI, DOM, DOL. Processors conforming to ISO-R1539-1972 require extensions to support these calls; namely, the FORTRAN processor must maintain association with the data block during execution of these subroutines by the executive routine. These calls accommodate executive routines which permit continuation of execution in the requesting program while the process input or output is being accomplished. These are related to the six standard calls as shown below:

STANDARD	EXTENSION
AISQW	AISQ
AIRDW	AIRD
AIW	AI
DIW	DI
DOMW	DOM
DOLW	DOL

wherein the arguments are defined identically as those of the corresponding standard calls.

For these extensions, the requesting program is required to have provision for periodically testing the status of the request. This is accomplished by use of the argument m.

All values returned by these extensions should not be considered as defined until such time as the parameter m indicates operation completed or error condition.

The process system designer must ensure the availability of these arguments to the process input/output interface subroutine in his total system. No method is given of how this will be achieved, although most existing industrial systems have features which can be used for this purpose, such as global common.

For the standard process I/O calls, the return to the calling program should not be made with m=2. On return m should equal 1 to indicate successful completion or m should be 3 or greater to indicate an error was found.

The argument j is defined to be processor dependent and can take several different forms. For example, in AISQW and AIRDW

1. An alias for the analog input similar to a logical unit number
2. A one word per analog input hardware address
3. A multiple word per analog input hardware address in which case the array j must be a multiple of the size of k.

B.3 Section 4 Notes

The basic unit of computer storage is the integer according to ISO FORTRAN. For the purposes of industrial computing it is necessary to view integer data as an ordered set of bits $(a_n, o_{n-1}, \ldots\ldots a_0)$ where the set is a place positional binary representation of an integer value.

For reasons of mathematical exactitude it is necessary to express the bit functions as a summation of a series, but on a "normal" two's complement binary computer, they will operate in the expected manner for such functions. However, on a binary coded decimal machine the results of these functions may not be in the expected manner for such functions and it will not be possible to reference every 'bit' of a word.

Items such as difference in word lengths, one's complement versus two's complement, negative numbers and BCD versus pure binary will affect transportability. However, due to the differences in the computers themselves, these differences are to be expected.
These functions are not intended to discourage the introduction of syntactic changes to FORTRAN.

B.3.1 Examples of Section 4

The following examples illustrate the use of Section 4 in less mathematical terminology. The processor is assumed to be a seven (7) bit +, two's complement machine, with the sign bit on the extreme left.

Let	j = 10	(0001010)
	m = 3	(0000011)

Then	IOR (j,m) = 11	(0001011)
	IAND (j,m) = 2	(0000010)
	NOT (j) = -11	(1110101)
	IEOR (j,m) = 9	(0001001)
	ISHFT (j,m) = -48	(1010000)
	ISHFT (j,-m) = 1	(0000001)
	ISHFT (m,j) = 0	(0000000)
	BTEST (j,m) = .TRUE.	
	IBSET (j,m) = 10	(0001010)
	IBCLR (j,m) = 2	(0000010)

B.4 Section 5 Notes

The format used for referencing time and date follows the ISO standard R2014-1971, writing of calendar dates in all-numeric form.

The standard ISA-S61.1 does not define the basis used or how the processor maintains date and time but does define how this information is made available to the program.

SECTION II

Proposed ISA Standard S61.2

INDUSTRIAL COMPUTER SYSTEM FORTRAN PROCEDURES FOR FILE ACCESS AND THE CONTROL OF FILE CONTENTION

Developed by the Industrial Real-Time FORTRAN Committee of the Workshop through ISA Standards Committee SC61. Presently under review by ISA Standards Review Procedures. As presented here it is substantially correct. The only likely technical change is an additional access privilege called PROTECTED READ. Approval by ISA is expected early in 1977.

PRECEDING PAGE BLANK-NOT FILMED

PREFACE

This forward, all footnotes and all Appendices are included for informational purposes and are not part of Standard ISA-S61.2.

This Standard has been prepared as a part of the service of the Instrument Society of America toward a goal of uniformity in the field of instrumentation. To be of real value this document should not be static but should be subjected to periodic review. Toward this end the Society welcomes all comments and criticisms and asks that they be addressed to the Standards and Practices Board Secretary, Instrument Society of America, 400 Stanwix Street, Pittsburgh, Pennsylvania, 15222.

The ISA Standards and Practices Department is aware of the growing need for attention to the metric system of units in general, and the International System of Units (SI) in particular, in the preparation of instrumentation standards. The Department is further aware of the benefits to users of ISA Standards in the U.S.A. of incorporating suitable references to the SI (and the metric system) in their business and professional dealings with other countries. Toward this end this Department will endeavor to introduce SI and SI-acceptable metric units in all new and revised standards to the greatest extent possible. The Metric Practice Guide, which has been published by the American Society for Testing and Materials as ANSI Z210.1 (ASTM E380-76), and future revisions, will be the reference guide for definitions, symbols, abbreviations and conversion factors.

The ISA Standards Committee on Industrial Computer System FORTRAN Procedures for Executive Functions and Process Input-Output SP61, operates within the ISA Standards and Practices Department, W. B. Miller, Vice President. The persons listed below served as members of this Committee:

M. R. Gordon-Clark, Chairman	Scott Paper Company
A. Arthur	IBM Corporation
F. E. Bearden	Modular Computer Systems
R. Caro	Foxboro Company
L. Cartright	Inland Steel Company
R. L. Curtis	ALCOA
W. Van Diehl	Hewlett-Packard
J. Froggatt	Westinghouse Electric Company
D. Frost	Honeywell, Inc.
M. Hands	Datum Inc.
C. C. Haskel	Union Carbide
T. L. Luekens	Johnson Service Company
O. Pettersen	Stanford University
W. A. Resch III	Tennessee Eastman Company
R. Signor	General Electric

Copyright 1976 by the Instrument Society of America

PRECEDING PAGE BLANK-NOT FILMED

1. SCOPE

This standard presents external procedure references for use in industrial computer control systems. These external procedure references provide means for accessing files, and also provide means for resolving problems of file access contention in a multiprogramming/multiprocessing environment. In a multiprogramming/multiprocessing environment, it is expected that concurrent programs will attempt to access the same file at the same time, therefore, the external procedure references defined in this standard provide the information necessary for the processor to resolve such simultaneous access in an orderly manner. The method for resolution of access control is left to the processor.

1.1 Definition

File: A collection of related records treated as a unit. For the purposes of this standard, records are viewed as being of fixed length. Record storage and access are independent of the internal format of records.

1.2 Background

In computing systems, individual programs may have various relationships:

- Programs are executed sequentially.
- Several programs can be operating concurrently but with no shared resources.
- Several programs can be operating concurrently and these concurrent programs may share resources. A file can be such a shared resource.

Files exist in all computing systems and can have various attributes and features, such as:

- A file can contain data, programs, or catalogue information.
- There can be a variety of ways for file access such as sequential, direct, and stream.
- A file can be created or deleted by a program, by a system utility, or at system generation time.
- A file can have security attributes associated with the file for the purpose of ensuring file privacy.
- When a file is associated with a program, this association can be restricted by the processor for reasons of privacy.
- A file can be associated with a set of related concurrent programs and this association can be restricted to assure orderly resolution of contention problems among the concurrent programs.
- A file can be internal or external to a program.
- A file can reside on fixed or removable media.
- A file can reside on main storage or backing storage.
- Restrictions for reasons of privacy or contention may apply to a file or a component of a file such as records and data items.

2. INTERFACES TO ACCESS FILES

2.1 File System Environment For This Standard

In industrial computer systems, concurrent program operation with shared resources such as files is a common occurrence. This standard does not address all the areas of file management but is concerned with the problems that most commonly arise in industrial computer systems.

Table 1 shows both those features covered by the standard and those excluded; however the excluded features may affect the result of a request for association of a concurrent program to a file. Such restrictions on associations are processor dependent and are outside the scope of this standard.

The argument m, shown below, shall be set equal to or greater than two (2) in value for all instances in which the request is not accepted by the executive routine. Individual implementation may specify unique values of m within the allowable range to designate the specific reason for which the request was rejected.

2.2 Create

Execution of a reference to the subroutine CFILW shall establish, but not open, a named file. Files established by CFILW do not have any privacy attribute to restrict a concurrent program from accessing the files. The form of call is:

CALL CFILW (j,n_1,n_2,m) where:

j specifies the file
The argument is either: a) an integer expression
or b) an integer array name
or c) a procedure name
The processor shall define which of the above three forms are acceptable.

n_1 specifies the number of integers per record in this file. This argument shall be an integer expression.

n_2 specifies the number of records in this file. This argument shall be an integer expression.

m is set on return to the calling program to indicate the disposition of the request.
The value must be 1 or greater
1 - File successfully created
2 or greater - File not created
This argument shall be an integer variable name or integer array element name.

2.3 Delete

Execution of a reference to the subroutine DFILW shall remove a file from the file system. Any file created by the mechanism of Section 2.2 can be deleted by the execution of a reference to DFILW. A file currently open to another program

TABLE I

FEATURES AND ATTRIBUTES OF FILES

Included In Standard	Excluded From Standard
- Files whose contents are considered to be data.	- Files whose contents are not considered to be data by the accessing concurrent program.
- Files which exist on fixed media only or on removable media which are not removed.	- Files which exist on removable media which are removed.
- Files which reside in main storage or in backing storage.	
- Files which are external to a concurrent program.	- Files which are internal to a concurrent program.
- Creation and deletion of files by a concurrent program.	- Creation and deletion of files by a system utility or at system generation.
- The association of a file to a concurrent program for both system created and for concurrent program created files.	
- Restrictions on file access as applied to the file.	- Restrictions on file access as applied to a component of a file.
- The association of a file to a concurrent program irrespective of the method of access (direct, sequential or stream).	
- Read and write methods of access for direct access files only (sequential files are covered by standard FORTRAN ISO R1539-1972).	- Methods of file access except for direct access.
	- Attributes of a file for the purpose of ensuring file privacy.

cannot be deleted. The form of this call is:

CALL DFILW (j,m) where:

j specifies the file
The argument is either: a) an integer expression
or b) an integer array name
or c) a procedure name
The processor shall define which of the above three forms are acceptable.

m is set on return to the calling program to indicate the disposition of the request.
The value must be 1 or greater
1 - *File successfully deleted*
2 or greater - File not deleted
This argument shall be an integer variable name or integer array element name.

2.4 Open

Execution of a reference to the subroutine OPENW shall associate the specified logical unit by the program with a named file, and shall define the desired access privilege of that program to the file. The form of this call is:

CALL OPENW (i,j,k,m) where:

i specifies the logical unit number of which the file named by the argument j is referenced in the program. This argument shall be an integer expression.

j specifies the file.
The argument is either: a) an integer expression
or b) an integer array name
or c) a procedure name
The processor shall define which of the above three forms are acceptable.

k specifies the desired access privilege under which the program wishes to receive the file and is a declaration of intended use by the program. This argument shall be an integer expression.
The following values are defined:
1 Read Only - The calling program can read but not write; other concurrent programs can read and write.

2. Shared - The calling program can read or write; other concurrent programs can read or write.

3. Exclusive Write - The calling program can read or write; other concurrent programs can only read.

4. Exclusive All - Only the calling program can access the file.

m set on return to the calling program to indicate the disposition of the request.
The value must be 1 or greater.
1 - File successfully opened to the calling program
2 or greater - File not open to the calling program
This argument shall be an integer variable name or integer array element name.

Limitations

If the file is currently open to another program, the disposition of a possible request for a particular access privilege will be as follows:

- Read Only - Fails if another program currently has Exclusive All privilege, otherwise succeeds
- Shared - Fails if another program currently has Exclusive All or Exclusive Write privileges, otherwise succeeds
- Exclusive Write - Fails if another program currently has Exclusive All, Exclusive Write, or Shared privileges, otherwise succeeds.
- Exclusive All - Fails

Any attempt to open a file will be successful only if the file exists. If the file was created by a mechanism outside of the standard, the attributes given to the file at its creation may restrict the granting of an access privilege to the program.

2.5 Close

Execution of a reference to the subroutine CLOSEW shall end the program association of the specified logical unit with a named file. This association will have been established by the execution of a previous OPENW call. The form of the call is:

CALL CLOSEW (i,m) where:

i specifies the logical unit number. This argument shall be an integer expression.

m is set on return to the calling program to indicate the disposition of the request.
The value must be 1 or greater
1 - File successfully closed to the program
2 or greater - Non-performance
This argument shall be an integer variable name or integer array element name.

2.6 Modify Access Privileges

Execution of a reference to the subroutine MODAPW shall change the calling program's access privilege to a previously opened file without closing and reopening the file.

If the request for change cannot be granted, the previous access privilege remains in force. The form of this call is:

CALL MODAPW (i,k,m) where:

i specifies the logical unit number. This argument shall be an integer expression.

k specifies the access privilege desired. This argument shall be an integer expression.
The following values are defined:
1. Read Only - The calling program can read but not write; other concurrent programs can read or write.

2 Shared - The calling program can read or write; other concurrent programs can also read or write.

3 Exclusive Write - The calling program can read or write; other concurrent programs can only read.

4 Exclusive All - Only the calling program can access the file.

m is set on return to the calling program to indicate the disposition of the request.
The value must be 1 or greater.
1 - Access privileges requested is granted to the program
2 or greater - Access privileges before the request remains in force
This argument shall be an integer variable name or integer array element name.

Limitations

If the file is currently open to another program, the disposition of a request for a particular access privilege will be as follows:

- Read Only - Fails if another program currently has Exclusive All privilege, otherwise succeeds
- Shared - Fails if another program currently has Exclusive All or Exclusive Write privileges, otherwise succeeds.
- Exclusive Write - Fails if another program has Exclusive All, Exclusive Write, or Shared privileges, otherwise succeeds
- Exclusive All - Fails

If the file was created by a mechanism outside of the standard, the attributes given to the file at its creation may restrict the granting of an access privilege to the program.

3. INPUT/OUTPUT TO UNFORMATTED DIRECT ACCESS FILES

Various methods of accessing files exist such as "sequential" and "direct." Direct access is a method in which items of information are stored and become available independently.

This section of the standard provides for unformatted direct access to files. Access to sequential files is defined by ISO R1539-1972. In this standard, direct access files are considered to consist of fixed length records. These files are considered to be resident in some mass memory that is permanently attached to the computer. The length of a record in these files is defined in terms of a unit which is the amount of storage occupied by one integer.

3.1 Direct File Read

The execution of a reference to the subroutine RDRW results in the sequential transfer of one data record from a file. The file is treated as a direct access file for selection of the record. The calling program must have opened the file and must currently have access privileges.

CALL RDRW (i,j,k,n,m) where:

i specifies the logical unit number. This argument shall be an integer expression.

j specifies the record number of the record to be read. This argument shall be an integer expression.

k designates the first variable into which information is to be placed. This argument shall be a) an integer variable name or integer array element name or b) integer array name.

n specifies the maximum number of integers that can be transferred.

m is set on return to the calling program to indicate the disposition of the request.
The value is 1 or greater
1 - Data transfer completed successfully.
2 or greater - Data transfer fails.
The argument shall be an integer variable name or integer array element name.

3.2 Direct File Write

The execution of a reference to the subroutine WRTRW writes unformatted direct accessed information into files which have previously been opened and access privileges assigned to the calling program. The request will only be successful if the program's access privilege is shared, exclusive write or exclusive all.

The forms of the calls are:

CALL WRTRW (i,j,k,n,m)

This call is identical to RDRW except for the direction of information transfer.

APPENDICES

These Appendices are not part of the ISA Standard S61.2 but are included to facilitate its use.

APPENDIX A

Considerations leading to Industrial Computer System FORTRAN Procedures for File Access.

A.1 Historical Development

This standard is a direct outgrowth of the International Purdue Workshop on Industrial Computer Systems whose goals are:

To make the definition, justification, hardware and software design, procurement, programming, installation, commissioning, operation and maintenance of industrial computer system more efficient and economical through the development of standards and/or guidelines on an international basis.

The Workshop formed several committees to achieve its objectives. The Fortran Language Committee was charged with the task of preparing a set of Industrial Process standards compatible with FORTRAN as defined by ISO R1539-1972*.

This standard is the result of that committee's work.

A.2 Criteria Used in Developing FORTRAN Standards

The committee assessed the status of FORTRAN as used in the industrial environment and followed the guidelines below in the development of the standards:

(1) The Standards should cover features commonly used by existing industrial computer systems.

(2) The standards should be easily implemented by most vendors.

(3) The standards should follow the syntax and intent of FORTRAN as defined by ISO R1539-1972.

(4) The standards should not restrict the future evolution of FORTRAN.

The development of FORTRAN standards is presently the responsibility of ANSI/X3J3. In order that ISA standards comply with the ANSI standards as far as possible, external-procedure references were used rather than direct changes or additions to the syntax of FORTRAN. This does not imply that this is the only way to provide these features, nor does it exclude the possibility or desirability that ANSI will develop the language syntax to perform these and other related forms.

*ISO R1539-1972 is the same as ANSI X3.9-1966.

APPENDIX B

NOTES BY SECTION

B.1 Section 2 Notes

This standard is a permissive standard in that it does not prescribe how the executive will respond to the external procedure references nor how the information on the access privileges to the file will be handled by the executive routine. Interchangeability of programs among different processors with different executive routines is reduced by this lack of prescription but the committee feels that it is premature to standardize to the degree necessary to achieve this interchangeability.

This standard does not cover all areas of file management. In particular no attempt is made to consider file privacy. In an industrial computer, system privacy is not a significant problem because the intent is to provide a common data base on plant operation for control and information purposes for all users of the system but the problem of contention is acute because of the asynchronous nature of the requests to the system.

File management can be very complex especially with removable media because the system must check that the correct media is present for all assess to the file. Such checking involved considerable system overhead which is much reduced if a restriction is made to non-removable media. The committee considers the restriction to non-removable media satisfactory for industrial system, especially if adequate measures are taken by the operators of the equipment at those times when removable media are changed.

B1.1 Example

Consider a file called FILNAM which contains data on a process. The file is eight (8) records long, each record contains ten (10) integers. The first seven (7) records contain process information, the eigth record contains status information.

Program ABC reads the process data in the first seven records of FILNAM and sets the process data in the first seven records of FILNAM and sets up the status information on the eighth record.

Program XYZ reads the status record of FILNAM, performs some actions, and resets the status record.

Program ABC is executed asynchronously when the process information changes. Program XYZ runs at regular intervals.

In this example the file contention procedures are used to ensure an orderly use of the status information of the file FILNAM.

```
C     example program ABC

      INTEGER FILNAM
      DIMENSION IVAL(10), JVAL(10)

C     open file logical unit number to
C        file name filnam
C        access privileges read only

      CALL OPENW(10, FILNAM,1,M)
      IF(M-1)90,10,90

C     read record number 1

   10 CALL RDRW(10,1,IVAL,10,M)
      IF(M-1)90,20,90
   20 CONTINUE

C     continue processing data

C     change access privileges to
C        exclusive all
C        and update record number 8

   30 CALL MODAPW(10,4,M)
      IF(M-1)30,40,30
   40 CALL WRTRW(10,8,JVAL,10,M)
      IF(M-1)90,50,90
   50 CALL CLOSEW(10,M)
         STOP

C     error handling routines

   90 CONTINUE
```

```
C     example program XYZ

      INTEGER FILNAM
      DIMENSION KVAL(10)

C     open file logical unit number 7
C        file name filnam
C        access privileges exclusive all

   10 CALL OPENW(7,FILNAM,4,M)
      IF(M-1)10,20,10

C     read record 8 and update contents

   20 CALL RDRW(17,8,KVAL,10,M)
      IF(M-1)90,30,90
   30 CONTINUE

C     continue processing data

C     write the updated record

      CALL WRTRW(7,8,KVAL,10,M)
      IF(M-1)90,40,90

   40 CALL CLOSEW(7,M)
      STOP

C     error handling routine

   90 CONTINUE
```

APPENDIX C

Relationship of Standard to the Draft Revised FORTRAN dpX3J3/1976

C.1 Revised FORTRAN Standard

The ANSI X3J3 committee has been considering revisions to standard FORTRAN (ISO R1539-1972) and has released a draft proposed standard, X3J3dpANS FORTRAN, for comments and review. The changes, clarifications, deletions and additions made to R1539-1972 do not invalidate any portion of this standard. The revised FORTRAN consists of a full language and a subset and it is recommended that any supplier providing the language in the proposed revision should follow the procedures given in the next section. These recommendations could be invalidated if significant changes are made to the present draft dpX3J3/1976 before a new FORTRAN standard is approved.

C.2 Full FORTRAN dpX3J3/1976

The standard is compatible with this FORTRAN. However, Section 3 of this standard - read/write to direct access media - is covered in more detail with more features in the input/output section of dpX3J3/1976.

C2.1 Extensions to Full FORTRAN dpX3J3/1976

Section 2 of this standard can be implemented by an extension to the INQUIRE feature of dpX3J3/1976. The recommended method for such an extension is to use two (2) more keywords.

ERVAL = n where n is an integer expression identical to the argument "m" in Section 2.

PRIVLEDGE = n where n is an integer expression identical to the argument "k" in Section 2.

The actions performed by execution of a reference to the subroutines, OPENW and MODAPW, are provided by the INQUIRE with suitable key words.

C.3 Subset of FORTRAN dpX3J3/1976

This standard is compatible with the proposed standard subset of FORTRAN dpX3J3/1976. However Section 3 of this standard read/write to direct access media - is covered by the READ/WRITE of FORTRAN dpX3J3/1976.

SECTION III

WORKING DOCUMENTS OF THE INDUSTRIAL REAL-TIME FORTRAN COMMITTEE CONCERNING WORK ON PROPOSED ISA STANDARD S61.3, INDUSTRIAL COMPUTER SYSTEM FORTRAN PROCEDURES FOR MANAGEMENT OF PARALLEL ACTIVITIES

This Section presents two of the many working papers developed by the Industrial Real-Time FORTRAN Committee in studying all aspects of the subject of tasking or the management of parallel activities. A draft standard for this area is now under development by the Committee. Mr. Caro's paper appeared in the *Minutes of the 1976 Spring Regional Meetings* as Appendix A IV D, pp. 51-65, *Part II, Technical Appendices and Tutorials*. An earlier version of Professor Petersen's paper appeared in the same Minutes, Appendix A-III D, pp. 273-311, *Part I, Narrative and Technical Appendices*.

A PROCESS CONTROL
VIEW OF
TASKING

- OR -

FORTRAN REQUIREMENTS
FOR THE SEQUENCING AND
CONTROL OF PARALLEL
CONCURRENT PROCESSES

BY RICHARD CARO
THE FOXBORO CO.
FOXBORO, MASS.

PRECEDING PAGE BLANK-NOT FILMED

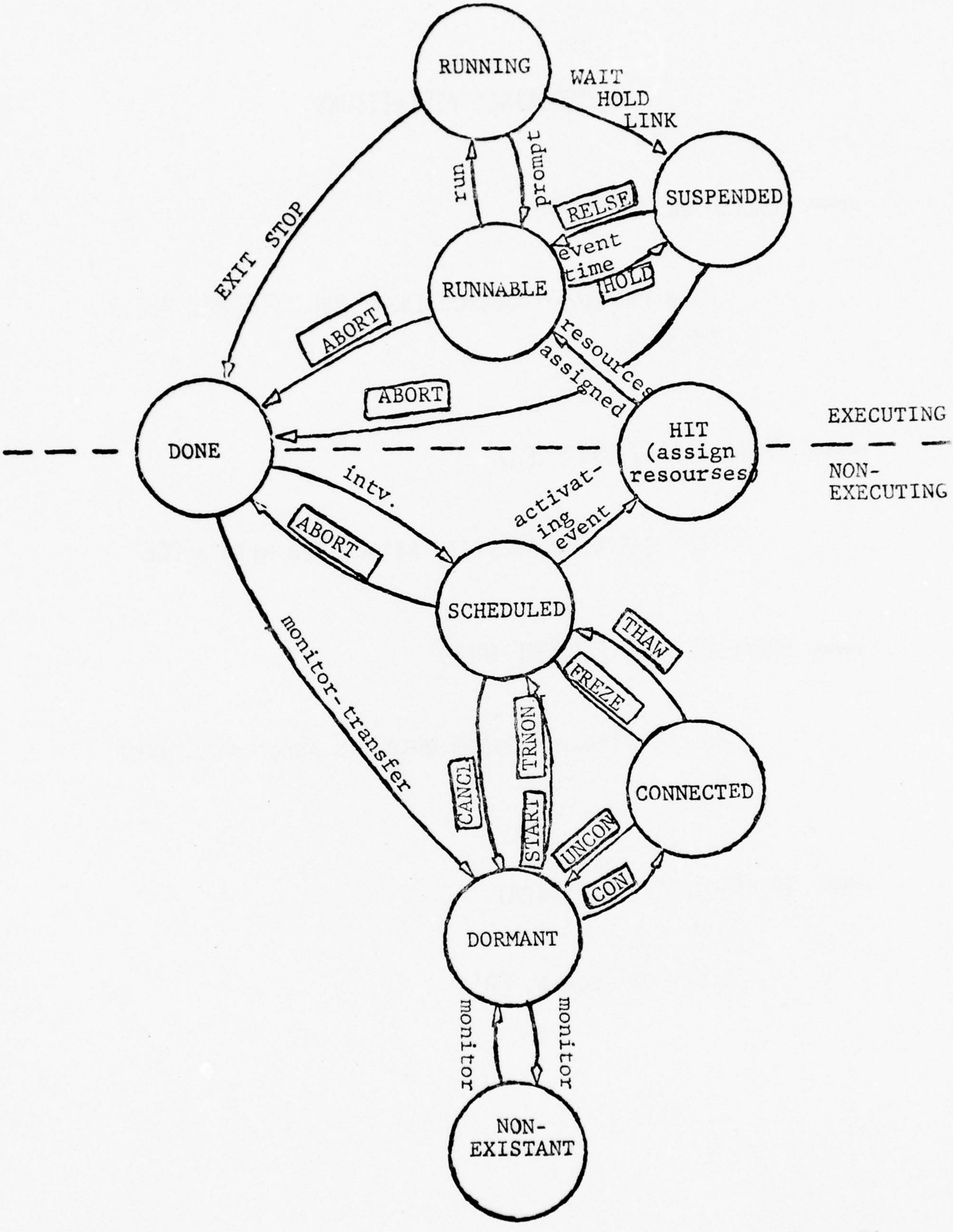

RUNNING
WAIT
HOLD
LINK
SUSPENDED
run
prompt
RELSE
event
time
HOLD
RUNNABLE
EXIT STOP
ABORT
resources
assigned
ABORT
DONE
HIT
(assign
resourses)
EXECUTING
NON-
EXECUTING
intv.
activat-
ing
event
ABORT
SCHEDULED
THAW
FREZE
monitor-transfer
CANCL
TRNON
START
CONNECTED
UNCON
CON
DORMANT
monitor
monitor
NON-
EXISTANT

FIGURE 1

STATE DIAGRAM

Dec. 74
Odd Pettersen

PRECEDING PAGE BLANK-NOT FILMED

"TASKING" DEFINITIONS

- EXECUTABLE PROGRAM

  MAIN PROGRAM + SUBROUTINES; REF. FORTREV 2.4.2 X3J3/71

- PARALLEL ACTIVITY (PA)

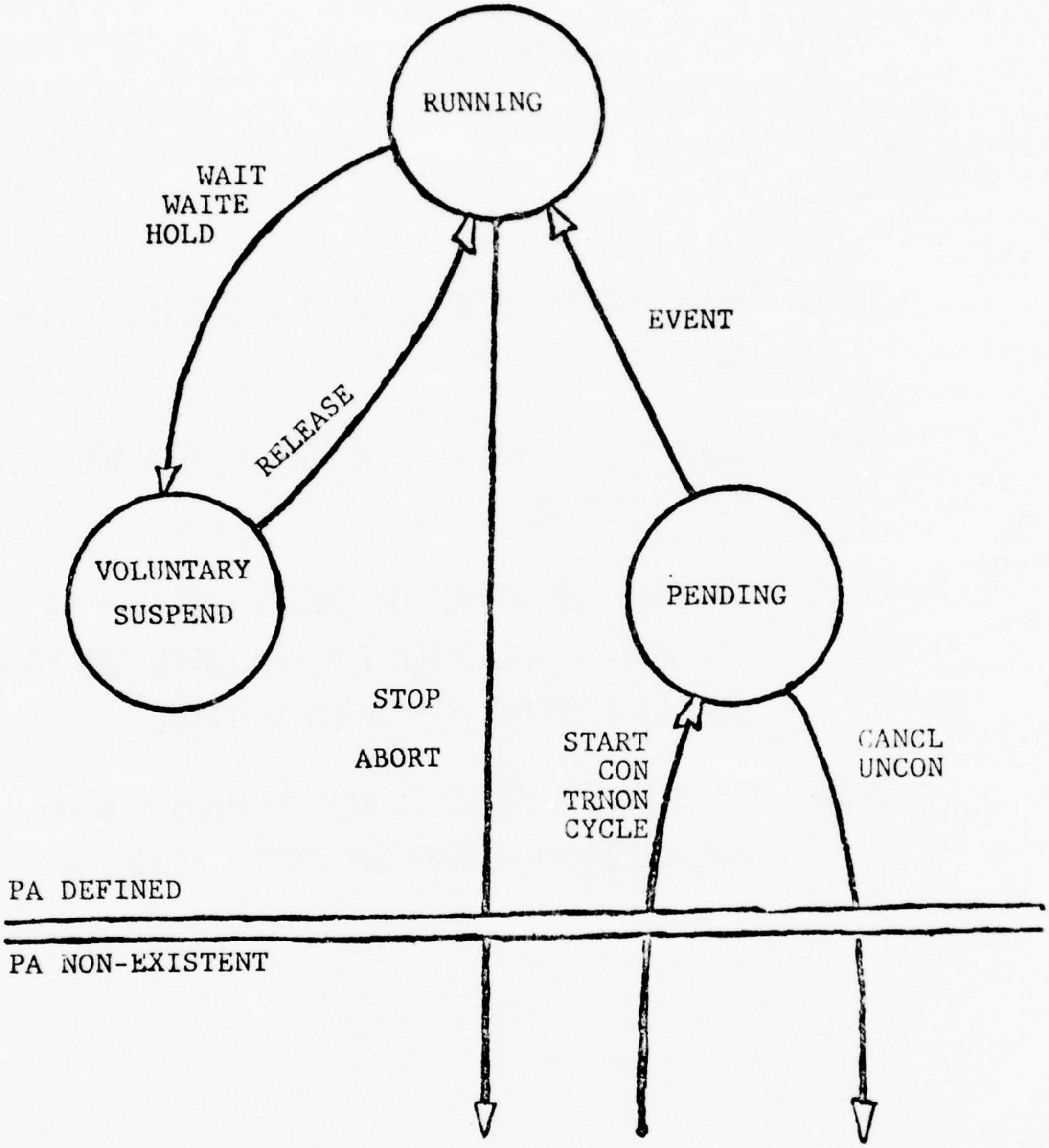
  TIME PARALLEL ACTIVITY ASSOCIATED WITH A PCE

- PARALLEL CODE ELEMENT (PCE)

  AN EXECUTABLE PROGRAM WHICH IS ASSOCIATED WITH A PA

- RELATED ACTIVITIES (RA)

  A SET OF RELATED PA'S


RUNNING
WAIT
WAITE
HOLD
RELEASE
EVENT
VOLUNTARY SUSPEND
PENDING
STOP
ABORT
START
CON
TRNON
CYCLE
CANCL
UNCON
PA DEFINED
PA NON-EXISTENT

PARALLEL ACTIVITY STATE DIAGRAM

FORTRAN PROCEDURES

SCHEDULING

TRNON	BEGIN EXECUTION OF A PA AT A SPECIFIED TIME OF DAY
START	BEGIN EXECUTION OF A PA FOLLOWING A SPECIFIED TIME INTERVAL
CYCLE	INITIATE THE SERIES OF EXECUTIONS OF A PA FOLLOWING A SPECIFIED TIME INTERVAL AND REPEATING AT A SECOND SPECIFIED INTERVAL
CANCL	DESCHEDULE A PA IF IT HAS PREVIOUSLY BEEN SCHEDULED AND REMOVE ANY CYCLIC REQUESTS

FORTRAN PROCEDURES (CONT'D)

- SEQUENCING

WAIT	PA EXECUTION IS SUSPENDED UNTIL THE SPECIFIED TIME INTERVAL HAS ELAPSED
HOLD	PA EXECUTION IS SUSPENDED UNTIL "RELEASED" BY ANOTHER PA
SYNC	PA REQUESTS PRIVELEGE TO ENTER A CRITICAL REGION
DSYNC	PA INDICATES END OF CRITICAL REGION
WAITE	PA EXECUTION IS SUSPENDED UNTIL THE SPECIFIED EVENT HAS OCCURED
STOP	PA IS TERMINATED IMMEDIATELY

FORTRAN PROCEDURES (CONT'D)

RELATED ACTIVITY CONTROL

RELESE	THE SPECIFIED PA WILL BE ALLOWED TO CONTINUE IF IT HAD BEEN IN A "HOLD" CONDITION
CON	THE SPECIFIED PA WILL BE EXECUTED IF THE SPECIFIED EVENT OCCURS
DCON	THE SPECIFIED PA WILL BE DISASSOCIATED WITH THE SPECIFIED EVENT
ABORT	THE SPECIFIED PA WILL BE TERMINATED AT THE NEXT AVAILABLE OPPORTUNITY

SYNTAX

CALL [START] (I, J, K, M)

WHERE:

I PA NAME (CHARACTER)

J THE TIME DELAY (INTEGER)

K UNITS OF TIME DELAY (INTEGER)

.EQ. 0 COUNTS OF SYSTEM CLOCK
.EQ. 1 MILLISECONDS
.EQ. 2 SECONDS
.EQ. 3 MINUTES
.EQ. 4 HOURS

M ERROR INDICATOR (INTEGER)
[VALUE DEFINED ON COMPLETION OF STATEMENT]

.LE. 0 UNDEFINED
.EQ. 1 REQUEST ACCEPTED
.GE. 2 REQUEST REJECTED
(2 → N MAY INDICATE REASON)

TASKING FUNCTIONAL REQUIREMENTS

☐ INITIATION

CAUSE PROGRAM TO BEGIN

☐ TERMINATION

CAUSE A PROGRAM TO CEASE EXECUTION

☐ SYNCHRONIZATION

ALLOW PROGRAMS WHICH SHARE RESOURCES TO TIME-PHASE COORDINATE EXECUTION

☐ COMMUNICATION

PASS DATA BETWEEN PROGRAMS

☐ EXCEPTION HANDLING

ALLOW USER TO SPECIFY SYSTEM RESPONSE TO DEFINABLE ERRORS

INITIATION

- PROGRAM NAME
- STARTING TIME
- OFF-SET FROM CURRENT TIME
- REPETITION TIME
- NUMBER OF REPETITIONS
- OVERRUN ERROR ACTION
- PRIORITY
- AUTHORITY
- PARAMETER LIST
- ERROR ACTION IF ALREADY INITIATED

TERMINATION

☐ SELF (PROGRAM ITSELF DECIDES)

- STOP NOW
- STOP DUE TO ERROR

☐ OTHER (CAUSE ANOTHER PROGRAM TO TERMINATE)

- NORMAL, AT CLEAN/SAFE OPPORTUNITY
- EMERGENCY, GET OUT NOW

☐ SERIES/SET (CAUSE A SET OF RELATED PROGRAMS TO ALL TERMINATE)

- NORMAL
- EMERGENCY

☐ GENERIC (CAUSE ALL SCHEDULED EXECUTIONS OF A SPECIFIC PROGRAM TO TERMINATE)

- NORMAL
- EMERGENCY

SYNCHRONIZATION

☐ WITH EVENT

- WAIT FOR EVENT
    - HARDWARE (CONTACT, INTERRUPT)
    - SOFTWARE (SEMAPHORE, FLAG, COMMON)

☐ WITH TIME

☐ RESOURCE - CRITICAL REGION

☐ WITH ANOTHER PROGRAM

- HOLD/RELEASE

☐ USE OF SHARED VARIABLE NAMES AND PROGRAM NAMES

COMMUNICATIONS

☐ SEND

- PROGRAM NAME (DESTINATION)
- DATA IDENTIFIER
- DATA VALUE
- DATA ROUTING
- WHEN (START TIME/OFFSET)
- ON EVENT
- AUTHORITY
- ERROR RESPONSE

☐ RECEIVE

- DATA IDENTIFIER
- DATA VALUE
- AUTHORITY
- ERROR RESPONSE
- ORIGIN PROGRAM NAME
- ACTUAL ROUTE

EXCEPTION HANDLING

☐ EXECUTION ERROR

- OVER/UNDER FLOW
- QUEUE FULL
- FORMAT MISSMATCH

☐ OVERRUN ON SCHEDULE

☐ LACK OF RESOURCES

☐ FAILURE OF RESOURCES

☐ PREDICTABLE DELAY

- DEVICE NOT READY
- LINK BUSY

GENERIC FORM OF CALL

```
CALL INITIATE  (PROGRAM NAME [PRIORITY = PRIORITY VALUE,]
        [AUTHORITY = AUTHORITY KEY,]

        [EVENT = EVENT MARK,

       [{TIME = TIME OF DAY      }]
       [{DELAY = TIME INTERVAL   }]
       [{NOW                     },]

       [ CYCLE = CYCLE INTERVAL,

        {FOR = TIME PERIOD
        {UNTIL = TIME OF DAY
        {FOREVER                ,

         LIST = PARAMETER LIST NAME
         CYCLE NAME = CYCLE NAME]
```

UNIVERSITETET I TRONDHEIM
NORGES TEKNISKE HØGSKOLE
INSTITUTT FOR TEKNISK KYBERNETIKK

THE UNIVERSITY OF TRONDHEIM
THE NORWEGIAN INSTITUTE OF TECHNOLOGY
Division of Engineering Cybernetics

RAPPORTNUMMER (REPORT NO.)
76-82-W

TILGJENGELIGHET (ACCESSIBILITY)
Accessible

RAPPORTENS TITTEL (TITLE) MANAGEMENT OF PARALLEL ACTIVITIES IN REAL-TIME FORTRAN State Model and State Transitions 2. edition	 DATO (DATE) October 1976 ANTALL SIDER OG BILAG (NO. OF PAGES) 46
SAKSBEARBEIDER/FORF. (AUTHOR) Odd Pettersen	ANSV. SIGN. (RESPONSIBLE) Odd Pettersen
OPPDRAGSGIVER PROJECT COLLABORATION INTERNATIONAL PURDUE WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS	PROSJEKTNUMMER (PROJ. NO.)

EKSTRAKT (ABSTRACT)

A working document within the framework of the development of ISA-standard S61.3, and written for the FORTRAN committees of International Purdue Workshop on Industrial Computer Systems. Two committees are active within this field: The American FORTRAN Committee of Purdue Workshop, and the FORTRAN Committee (TC1) of Purdue Europe.

The author has been actively enganged in the American committee for the past nearly 2 years and is now a member of the newly established European committee.

The 1 st. edition of the document has been discussed within TC1 of Purdue Europe and obtained approval, provided some minor changes were made, mostly of editorial nature. These changes have been incorporated into the 2nd. edition.

3 STIKKORD (KEYWORDS)

Real Time Programming
FORTRAN Standardization
Computer Control Languages

CONTENTS

Page

1. INTRODUCTION 51

2. DEFINITIONS OF IMPORTANT TERMS 53

3. MATHEMATICAL MODELS FOR PARALLEL ACTIVITIES 57

4. DESIGN OF A SUITABLE MODEL 58
    4.1. Multiple calls for state transitions 60
    4.2. Events 63
    4.3. Definition of states 65

5. DEFINITION OF ABSOLUTE TIME 66

6. DEFINITION OF SUBROUTINE AND PROCEDURE CALLS 67
    6.1. Summary of subroutine calls 68
    6.2. Creation of a new activity . . . . CREATE 72
    6.3. Eliminating an activity from the real-time system KILL 73
    6.4. Starting an Activity Immediately or after a Specified time delay . . . . START 74
    6.5. Starting an Activity at a Specified Absolute Time TRNON 75
    6.6. Starting an Activity in Periodic Execution 76
        6.6.1 Initial start immediately or after a specified time delay . . . . CYCLE 76
        6.6.2 Initial start at specified absolute time TCYCLE 78
    6.7. Connection of a program to an event . . . . CON 79
    6.8. Terminating periodic execution . . . . DECYCLE 80
    6.9. Elimination of an event connection . . . . DECON 81
    6.10. Elimination of previous scheduling . . . . CANCL 81
    6.11. Delaying Continuation of an Activity . . . . . . WAIT 83
    6.12. Synchronizing suspension of an activity 84
        6.12.1 Wait on Semaphore . . . . WTS 85
        6.12.2 Release of Semaphore . . . . SIGNAL 86
        6.12.3 Entry of Conditional Critical region . . . REGION 86
        6.12.4 Exit from Critical region . . . . REGEND 88
        6.12.5 Waiting for an event . . . . AWAIT 89
        6.12.6 An alternative definition of critical region call . . 90
    6.13. Normal termination of execution . . . . EXIT 91
    6.14. Status information of Pending-queue . . . . QPEND 91

REFERENCES 93

1. INTRODUCTION

This document represents a step of a continuous updating of working material within International Purdue Workshop on Industrial Computer Systems, Technical Committees on Real-Time FORTRAN. Primarily, it is written to express my own viewpoints, resulting from some years' work within the FORTRAN committees: First, with the American Committee, and more recently as a member of the European TC1 (Technical Committee on Real-Time FORTRAN).

The previous document in the series, ref. [18], has been distributed within the American as well as the European Committee on Real-Time FORTRAN. With some minor suggestions to improvements, the document was approved by the European Committee, at the committee meeting in Brussels, in the beginning of September, 1976. The present document represents a quick rewriting of [18], with an effort to incorporate the ideas and improvements suggested at the Brussels meeting. The document has been prepared before the Fourth International Meeting of the International Purdue Workshop on Industrial Computer Systems, which will be held on November 8. - 11., 1976. The short time has prevented a new discussion of the contents, within the European TC1. Since the predecessor of this document was largely accepted by TC1 with only minor requests for changes, however, the present document may be considered to express largely the viewpoints of the European Committee on Real-Time FORTRAN within "Purdue Europe".

It feels appropriate to remind about the short range goal for the FORTRAN committeework. As in previous documents, I consider it urgent to emphasize that the work on standardization of Real-Time mechanisms should be concluded at an early date. It is very important to keep as close to the current Standard FORTRAN as possible, however. Deviations should only be accepted in those cases where basic principles inherent with concurrent operation of real-time programs are not covered by the the current ANSI standard. Consequently, we encounter now a new situation, with the recent release by ANSI's X3J3 committee of their new "draft proposed ANS FORTRAN, X3J3/76". Although not finally approved as the new FORTRAN standard, one must expect the new standard to be quite close to the present draft. This will influence our work on real-time mechanisms for FORTRAN quite considerably. Time has not permitted the author of the present document to fully incorporate all obvious modifications caused by the new standard. It is also considered a subject for future committee work to consider all consequences of the new standard. Consequently, the reader might find portions of the present document obviously written at an early stage while other portions are of more recent date, influenced by recent committee discussions as well as the draft proposed FORTRAN. I hope that the reader will understand the reasons for this lack of homogeneity and that the main principles and ideas will be evident, despite some inhomogeneities in the forms of the document.

During the spring of 1975, the US FORTRAN committee achieved some significant results in the handling of Parallel Activities, formerly termed "tasking". Valuable contributions are also made by members of the European TC1. The great number of valuable contributions during discussions has not been matched by written papers as documentation of the orally achieved results, however, at least within the framework of Purdue Workshop. Like [18], the present paper represents an attempt to satisfy some of these needs. Additionally, the present document benefits from a somewhat higher degree of approvement from the European TC1 than [18] does. Consequently, I will suggest, that the present document is considered as a basis for further committee work and that the consequences of the draft proposed ANS FORTRAN will be discussed, so that its influence on the real-time mechanisms can be revealed and eventually incorporated in a Standard paper on Real-Time FORTRAN, hopefully attainable in a near future.

2. DEFINITIONS OF IMPORTANT TERMS

Throughout the present paper, the following terms and their definitions are used. Some of the definitions correspond largely to those formulated at the committe meeting during the second American Regional Meeting of Purdue Workshop in April 1975. To some extent, the definitions are inspired by [4].

The terms are written in capital letters in their definitions. Terms defined elsewhere in this vocabulary are underlined. It will, however, be underlined only at the first occurrence within the same definition paragraph.

OPERATION.	A deterministic rule for the generation of a finite set of data from another finite set of data.
COMPUTATION.	A finite set of operations applied on a finite set of data, in an attempt to solve a problem. If the computation really solves the problem, it will also be an ALGORITHM.
ACTIVITY.	A computation, where the operations are performed in a strict sequential order.
PARALLEL ACTIVITIES.	A set of activities whose operations may overlap in time. Parallel activities are DISJOINT if every one of them only refers PRIVATE DATA, i.e. data that are not accessed by any other activity of the set. They are INTERACTING if they refer to SHARED DATA, i.e. data that are accessed by more than one activity of the set.
PROGRAM.	A description of a computation, expressed in a formal language, PROGRAMMING LANGUAGE.
SEQUENTIAL PROGRAM.	A program consisting of operations that can only be performed strictly one at a time without any overlap in time. See definition of "Activity". An Activity is described by one or more sequential programs excluding each other in time.
PHYSICAL PROCESSOR.	A physical component capable of executing an activity defined by a stored program.

VIRTUAL PROCESSOR.	A simulated component, capable of executing an activity defined by a stored program. The virtual processor may be realized by a physical processor or simulated by program execution.
MULTIPROGRAMMING.	Programming techniques or organizational forms which make use of, or allow, parallel activities.
SYNCHRONIZATION.	A general term for restrictions regarding the order in which operations are performed. E.g. a synchronization rule may specify the order, priority or mutual exclusion in time between two or more operations. Specifically, the term synchronization will here be tied to parallel activities. According to the definition above, only interacting parallel activities are relevant to consider for synchronization.
CRITICAL REGION.	A part of a sequential program operating on shared data such that this program part must have exclusive access to the shared data during the execution.
SEMAPHORE.	A structure of shared data, used for the exchange of synchronizing signals between interacting parallel activities. Operations on a semaphore must only be performed within a critical region.
MONITOR.	A structure of shared data and a set of operations on this structure, for the purpose of synchronization of interacting parallel activities. The shared data can only be accessed through appropriate activations of the set of operations contained in the monitor. In this respect, the data are local to these operations, while the operations are shared. The operations can only be activated by one of the controlling parallel activities at a time.
MESSAGE BUFFER.	A structure of shared data, used for exchange of data between parallel activities.

OPERATING SYSTEM.	An organized collection of system programs acting as an interface between computer hardware and users, providing users with a set of facilities to simplify the design, coding, program error discovery, and maintenance of programs, as well as controlling the allocation of resources to assure efficient operation [14]. See executive system.
EXECUTIVE SYSTEM.	The part of an operating system controlling the allocation of resources in a computing system. One major resource is processor execution time[*].
SUSPENDED.	One of the definite states of an activity. A suspended activity has stopped the execution of its virtual processor, and is waiting for a specified event to continue the execution of its virtual processor.
CONDITIONAL CRITICAL REGION.	A program structure providing synchronization mechanisms between interacting parallel activities. A conditional critical region is a critical region realized as a call to the executive system, comprising specifications of conditions for the calling program's entrance into the critical region. If such entrance permission is not granted immediately, the calling program will be suspended.

---

[*] The author is aware of the flourishing number of diverging definitions within this subject. Frequently, operating system is used where this paper uses executive system. The determinant reason for the choice in this paper is an effort to find a term less biased with fixed but differing interpretations. Since operating systems often include system programs providing other facilities (like those listed above), there is a need for a term with a more restricted scope, and also for one that is less tied to specific but somtimes conflicting meanings.

EVENT.	A significant discrete occurrence or incident which is intended to affect some program execution in a planned manner. The source of an event can belong logically to an entity distinctively apart from the affected program unit or units. An event itself occurs instantaneously and is of infinitesimal duration. The fact that an event has occurred is indicated by an EVENTMARK which is a binary-valued program variable of type Boolean.

It is worth noting, that the term "Parallel Code Element", suggested at the Workshop at Purdue April '75, should be completely superfluous. "Code" is covered by "program", and "parallel" should pertain to "activity" rather than the programmed realization of it.

Please also remark, that I have found it logical and linguistic correct to use the term "activity" where the committee discussion ended up with "parallel activity". It is "parallel" only when "parallelled" with some other activity. Logically, one could have chosen "parallelable", indicating that the activity might be activated in parallel with some other, similar activity. The last term seems rather cumbersome and artificial,however, and I see no real need for the emphasis of the "parallellable" quality, since "Parallel Activities" are defined specifically.

Obviously, there is a need for a term covering several activities performing in parallel, and now the term "parallel activity" fits very well. At the committee meetings, the term "set of parallel activities" was proposed; this will now appear unnecessary complicated in comparison.

3. MATHEMATICAL MODELS FOR PARALLEL ACTIVITIES

The clarity and distinction in the description of an activity's various states and transfer between states, is greatly enhanced through the use of "state diagram" and state transitions between states. This idea was recognized early, and several different state diagrams have, through the years, been proposed in the discussion within the Workshop. The state diagram, with distinct states and transitions between states is,in fact, not only a practical descriptive method for clarification, but it yields a mathematical model of the system, with close relation to automata theory.

Theoretically, any discrete, simple sequential machine can be represented by a model, which can be of form of a so called Mealy or a Moore model, which are theoretically equivalent. Both models are characterized by a finite set of distinct states and a finite set of transitions between the states.

Given any initial state, an"input" may cause a transition to another state. Comparing different initial states, the same "input" may cause transitions to different states, but for any initial state, the transition for a given input is non-ambiguous. In a Mealy model, output functions are linked to the transitions, and in a Moore model, the output functions are associated with the states. For the discussion of our topic, the Moore type appears the most suitable, and "output" may be interpreted as "actions" in the state, for example program execution. The Moore and Mealy models are defined for single sequential processes, i.e. the process (activity) is in one and only one state at any instant of time. The concept is, however, very easily extended to a multiple of separate activities:

Such a system can simply be considered as modelled by a number of disjoint but similar diagrams. It seems feasible to apply a three-dimentional picture, with the similar individual diagrams sandwitched on top of each other and oriented so that the identical states of the individual diagrams cover each other.

The total system will have a number of state "stacks", the same number as the number of states in each diagram. Each "stack" represents the particular state of all activities, and it is easy to impose certain restrictions to the maximum number of activities entering the same state: In the composite model, this will be the maximum number of "entered" wafers in the stack. For the system, the limitations are dictated by available resources, which, for example, can be table space.

In designing a suitable state model, several possibilities will generally exist, and several of these can seem equally advantageous. All feasible models should, however, exhibit the following features, corresponding to requirements for the system:

1. The transitions must be non-ambiguous (as discussed above).
2. The transitions are carried out instantly, i.e. in zero time.
3. The model should not contain terminal states.
4. The model should be a Markov system, i.e. transitions from a state should depend only upon the state and current conditions, not on the previous state and the transitions that brought the activity into the present state.
5. An activity should exist in one state only at a time.

A terminal state is a state with transitions only originating in the state and none conducting into it, or transitions only conducting into the state and none going out from it. The reason for this requirement is obvious: It requires transitions, outside the model to bring an activity to or from such a state.

4. DESIGN OF A SUITABLE MODEL

It is not immediately evident, that the set of models, feasible from a user's point of view, is the same as the set of feasible models as seen from the operating system. The sets will probably overlap, however, indicating that it should be possible to find a model, suitable from both viewpoints.

If we set the goal to find an *optimal* model, however, it becomes less likely that a model, satisfactory from both viewpoints,can be found.

During the years, several models have been proposed, evidently attempting to satisfy diverging requirements from the user's and the operating systems's needs. It is probably safe to say, that compromises have been necessary and thus the model has never been optimal, from either viewpoint. Let me , shortly, explain certain controversial requirements:

- The operating system is concerned with assigning processor to an activity, ready to execute. Usually, there are more activities than available processors; thus some timesharing must be performed. A model must satisfy this and must contain states like RUNNING, RUNNABLE etc. The user, on the other hand, basically doesn't want to bother with these problems, on which he does not have any influence anyway. His requirement is just to get his activity executed within a tolerable time limit.

- As seen from the operating system, memory and other resources must be made available before an activity can execute. On the other hand, in order to economize the use of limited amount of resources, these may frequently be shared. For example is it sometimes necessary to release some memory space from a preempted program (core swapping). The order, in which resources should be made available to competing activities may be the object of considerable evaluation, compromises and complex solutions. It is not very likely that a user wants to, or should be allowed to, bother with these detailed problems, belonging to the operating system's area of responsibility.

The model proposed in the following is intended to satisfy the requirements sketched above and in the previous paragraph, optimized from the user's viewpoint. It is illustrated by the transition diagram of Figure 1 and conforms to the model, unanimously attained during the FORTRAN committee meeting in Phoenix, January 1975.

The following conventions are used for the drawing and symbols, adopted from [5] and also used in my preceding paper [1]:

State transitions are effected by subroutines and procedures identified by standardized names, by executive operations and by various events. These categories are differentiated between in the diagram:

- Capital letters in box: Effect on an activity imposed by another activity. I.e. a subroutine call in one activity has the indicated effect on another activity.

- Capital letters without box: Effect imposed by the activity itself.

- Small letters: Executive operations performed by the monitor, possibly triggered by an event.

The transitions are largely identical with those of [1]. I have found it valuable to adopt from [5] another transition for cyclic operation: TCYCLE. Further, there are some changes in synchronizing transitions. These changes will be explained in later paragraphs.

4.1. Multiple calls for state transitions.

One must accept that several different programs(activities) may issue conflicting transition calls regarding one and the same activity. Thus, it becomes necessary to clarify the situations that may arise, analyze the possible precautions, and to decide how these problems should be resolved.

Let us,firstly, mention two typical cases which occur commonly:

a. An activity may be scheduled by TRNON or START for execution at some future time, but also connected to several other events,for instance by call of CON.One of these could,for example, be the activation of a button on the operator's console, signalling the activity to execute.

b. A running activity may want (need) to schedule itself, by call of START, TRNON or CON.

Rule no. 5 of para. 3 denies an activity to be in more than one state at a time. This rule seems to be agreed upon as necessary, by all committee members, besides it is a fundamental condition for keeping the model in accordance with theoretical models of automata theory. Let us, therefore, try to explore all possible, as well as impossible, solution methods:

1. Rule 5 is abandoned, thus permitting the activity to be in more than one state at a time.
   This is not acceptable, from arguments above and several others, expressed by other committee members during discussions.

2. Reject any further calls for activation of an activity; i.e. reject calls that conflict with the present state.
   Discussion of consequences:
   It is indeterminate which of several activating activities that wins the race of acquiring the object activity the first time. Thus, the losing activities must be informed correspondingly, to enable them to repeat the call until success. This repeated calling must be programmed, in the user's program, and involves very much overhead, unnecessary

because it is inherently linked to this method, as opposed to the method discussed in section 3 below.

Another deficiency of this method is that the inclusion of the program statements for the repeated calls, in the application program, represents an extra workload on the application programmer.

The method makes the operating system implementation somewhat simpler, though. The author is, however, not inclined to emphasize the importance of simple implementation of operating systems, at the expence of run-time overhead as well as convenience and safety of application programming.

3. Call to the executive system (by calls of START, TRNON or CON) for state transition to PENDING causes listing in a table, appropriately called "pending-table". Each activity may have a maximum of one entry in this table, but complex conditions should be allowed: Limited only by some maximum table size, all valid subroutine calls, intended to cause state transition to PENDING should have its "running-condition" OR-ed to the event-function constituting the condition for the subsequent state transfer to RUNNING. This rule, besides being quite obvious, has a further advantage: It gives a very simple and logical operation of the CYCLE, TCYCLE, and CON calls. i.e. the calls causing possibility of multiple subsequent activations (RUNs): Whenever an activity attains state DORMANT, (by execution of EXIT), the pending-table is checked. If listed, the activity will be transferred immediately to state PENDING, guided by an OR-function of run-conditions from the table-entry. If the listing call was one of the recurrence calls CYCLE, TCYCLE or CON, the table entry will remain. The same will be the case, if multiple calls of START or TRNON were made, leaving the future activation times in the table. Thus, there will be no difference, in principle, between the immediate effects of multiple calls of START or TRNON and the recurrence calls CYCLE, TCYCLE, and CON. The effect of the recurrence calls will only be, that the remaining conditions in the table entry will be adjusted recursively, at the instant when the activity is transferred past DORMANT to PENDING after an EXIT.

4. (Applicable if the standard and the implementation allow entirely new activities to be created dynamically:)

Any new activation causes the creation of a new activity, similar or equivalent (or even identical) to the previous one. The newly created activity must get its own resources, share of resources, or requirement for common resources. This includes its own table space in the primary store (core store).

5. A combination of methods no. 3 and 4 above:
Different (parent) activities create or activate different identical child activities. Several activations of the same named activity from the same parent activity, however, refer to the very same child activity, such that these succeeding transition calls should merely be listed (queued).
From a standardization point of view, method no. 5 looks preferable: It is advantageous because of its generality; it is a superset of methods 3 and 4. Thus, it is convenient in the standard to leave it to the particular implementation, to what extent new activities may be created. This method also simplifies the resolution of the possible ambiguity relating to which activations of an object activity that shall be eliminated by a call of DECYCLE, DECON or CANCL. It seems reasonable to let these calls, from a "parent" activity, affect only its child activities, i.e. activities created by the same activity.

It is imperative to mention a word of caution, however, against an impetuous acceptance of recursive activity creation as a permissible feature in the standard. Several reputed authors have warned against the implementation of this feature (see for example [15], [16]). The main reason is the considerable amount of added complexity of the operating system which again may reduce software reliability. Related to this question, could also be mentioned several authors' questioning of the necessity and desirability of recursive subroutine calls. See for example D.E. Knuth's well known and rather recent article [17]. Since it is warned against recursive functions in pure sequential programs, it seems even more important to be sceptic, dealing with real time concurrent programs.

Another strong argument for rejection of recursivity is the fact that we are talking about FORTRAN, not an LTPL. Let us be realistic enough to acknowledge the short-time goal of getting to a conclusion in the work of standardizing Real-time FORTRAN. I am very much concerned, that an inclusion of dynamic and recursive creation of activities involves the danger of reducing software reliability as well as stretching the date of a final Standard S61 far out in the future.

Concluding, I find method no.3 above to be the best among those discussed, for the handling of multiple calls for state transitions.

4.2. Events.

Crucial to the description of the model is the term EVENT, defined in paragraph 2. Some additional explanations may be appropriate:

> The occurrence may be external or internal: The effect of an external event will be transmitted through the processor's input interface system as an interrupt request signal or a signal actively read by some program statements. The source of an external event will generally be of some physical nature.
>
> An internal event is characterized by some condition change within a program as a consequence of some program action. With relation to the effect of an event, it is not distinguished between external or internal nature of its origin.

As evident from the definition, *time* can be a basis for an event. A prevalent time is defined, related to the equation:

$$B \leftarrow t > t1 \qquad (1)$$

where:

t is wall clock time

$t1$ is some predetermined point of time

B is the eventmark, recognized as a binary boolean variable.

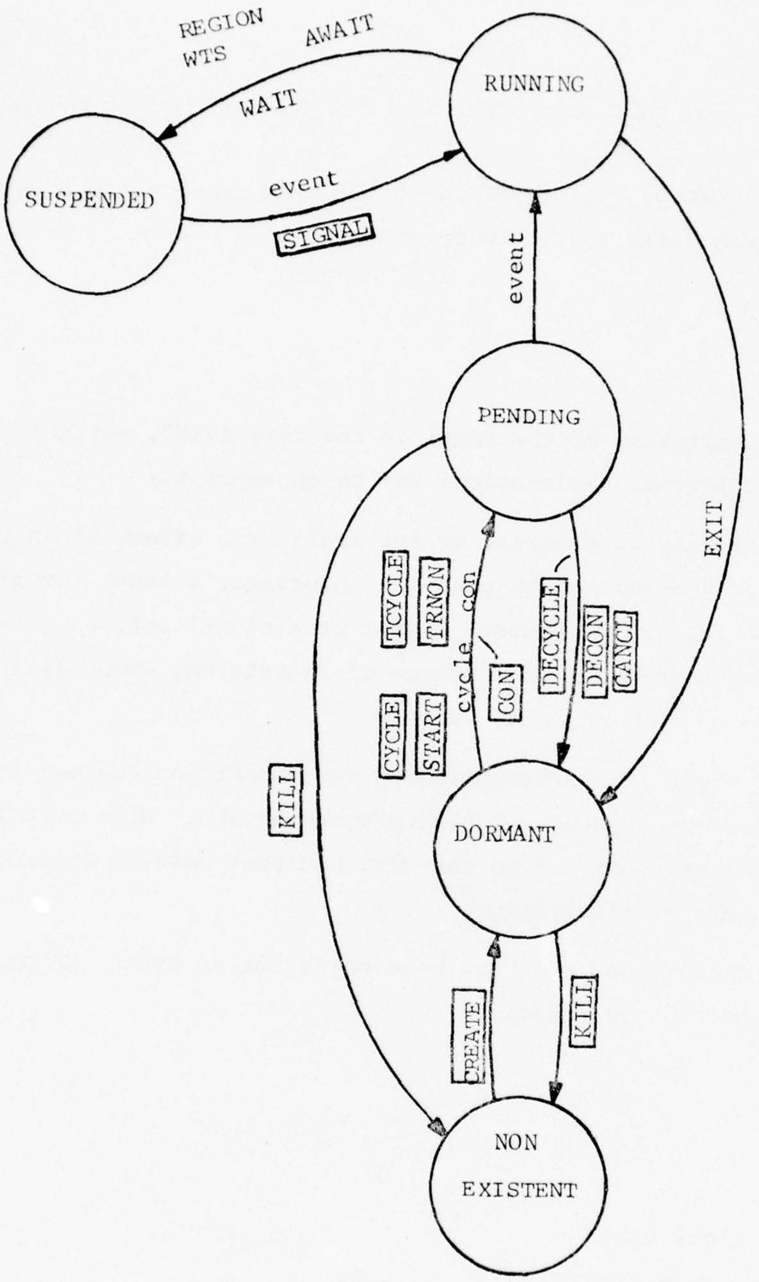

REGION
WTS
AWAIT
WAIT
RUNNING
SUSPENDED
event
SIGNAL
event
PENDING
EXIT
TCYCLE
TRNON
cycle con
CON
DECYCLE
DECON
CANCL
CYCLE
START
KILL
DORMANT
CREATE
KILL
NON
EXISTENT

Figure 1. STATE AND TRANSITION DIAGRAM

4.3 Definition of states.

As illustrated in Figure 1, the model has 5 states, defined by the committee as follows:

NON-EXISTENT

A non-existent activity is unknown to the activity manager.

DORMANT

A dormant activity is known to the activity manager and is not in any of the states PENDING,RUNNING, or SUSPENDED.

PENDING

A pending activity has been associated with an event such that when the event occurs, the activity will be transferred to state RUNNING.

RUNNING

A running activity is executing in its virtual processor. This means that its program will be executing on a real processor if all necessary resources are assigned to it. Otherwise, it will be waiting for its required resources.

SUSPENDED

A suspended activity has stopped the execution of its virtual processor, and is waiting for a specified event to continue the execution of its virtual processor.

5. DEFINITION OF ABSOLUTE TIME

Time must be defined accurately and unambiguously, for example when used as parameters for TRNON and similar subroutine and executive calls. As explained in [1], ambiguity results from modular interpretation, if higher order components and zero reference point of the complete time description are omitted. The only remedy is to include all these components (hour, day, month, year, reference time zero, eg. AD), either explicitely as contents in table elements, or implicitely as generally acknowledged constants.

Usually, the highest order components, like month and year, as well as the zero reference, are implicitely acknowledged. This works usually fine, as long as all references refer to the same (current) module. Ambiguity results on the boundary between modules, however. An example will clarify:

On March 30., an activation call for a certain activity is issued, by CALL TRNON. Activation time is specified to be some time on date 15. Is this to be interpreted as the 15th of next month, i.e. April 15., or are we just very late and really mean March 15.? If so: should the activity be activated immediately?

The only safe and correct remedy is to include all time components, at least in principle. This does not mean that the user always has to specify all components. All components should be included in a referred array, however, making the user responsible for appropriate modification of higher order components on the boundary between two time modules. In the example shown above, "March" was probably the "old" contents of the month-element of the time spesification array. It should be the user's responsibility to change this to "April", if April 15. is the expected time of activation.

This makes it necessary, wherever absolute time is specified, to provide for table space for all components of the complete time specification. The only elements that may be omitted are the time zero and whether time refers to "local time" ("Normal" or possibly "Daylight savings time") or UTC (GMT). But the omission of these elements must indeed presuppose that they are acknowledged implicitely.

The order, in which the elements are arranged in the tables, should indeed be monotonic. I have heard considerable arguing for a decreasing order:hour-minutes-seconds etc. for increasing element numbers. In [18], I capitulated for these arguments and arranged the time elements in a non-monotonic order. After discussions at the Brussel meeting in Sept. '76 of the European TC1, I have rearranged the order again, back to my earlier proposals, as contained in [1].

All technical and logical reasons favor the following order:

Index 1 Basic time unit
" 2 second
" 3 minute
" 4 hour
" 5 day
" 6 month
" 7 year

People arguing against this order, in favor of reverse order of index 1 - 4 only, are herewith invited to defend their position with convincing references to "already standardized" authoritative sources. I must admit, that I have been unable to recover any reference to such sources. *)

6. DEFINITION OF SUBROUTINE AND PROCEDURE CALLS

An activity may make a state transition, by virtue of a monitor call within the activity's own program or another program, or by reason of an event previously specified by the activity itself or some other activity. In the state diagram, Fig. 1, transitions caused by monitor calls within another program are indicated by capital letters in "box"; transitions caused by direct system calls from own program are indicated by capital letters without box. Finally, transitions caused by som different event are indicated by small letters. Incidentally, there is basically no important reason for emphasizing between time events and other events. A "time event" is similar to any other event (see definition): The occurrence of wall clock time exceeding a predetermined time, compared with, for example, some other quantity exceeding some predetermined limit value. Consequently, one could have defined a unified set of transitions, and they could be dependent upon a time event or some other event, according to the specific needs.

---

*) I haven't been motivated either; on the contrary, I have continuously been opposed to a non-monotonic order.

There are some basic difficulties in FORTRAN, however, of expressing general events in simple ways. I have,thus, found it necessary, for those subroutines dealing with general events, to define the event specification simply as a reference to an array, where all necessary specifications are found. This array can hardly be standardized at the present time; thus it must be left processor-dependent. Consequently, the specification of general events is awkward, compared to how time events can be specified. This justfies the use of separate procedures for time events: these are in majority, and they are probably used most often; the specification of time events should therefore not be unduly complicated.

The set of subroutine calls included here is believed to be complete for state transitions within the present state model. Subroutines like CHNGE and STTSK, described in [3], are not included, since they are not linked to state transitions. It should be considered whether these, or similar subroutines, should be added to the set. Some of the subroutine calls, included in [3] and [6] are not included because they should be superfluous in the present model. Among the subroutine calls vanishing are HOLD and RELSE. These are substituted by the more general semaphore calls WTS and SIGNAL;the WTS(WaiT on Semaphore) is identical with Dijkstra's P, while SIGNAL is identical to the V semaphore function. Three other calls are also available for synchronization purposes: REGION, REGEND and AWAIT; the first two of these are given new and more obvious names but are otherwise identical to SYNC, SYNCEND in [1]. Synchronization is explained further in the introductory part of paragraph 6.12. ABORT is not included because it has been considered dangerous, according to discussions and opinions within the committee. If found appropriate or necessary, from succeeding discussions, an ABORT call may easily be appended. Personally, I do not find it necessary to get involved in that discussion at the present time.

6.1. Summary of subroutine calls.

This paragraph contains a summary of the subroutine calls described in the following paragraphs.

The following terms and designations for parameters apply to several of the calls. If the exact meaning of these parameter designations deviates from what is defined below, it will be marked specifically in the detailed description of the call.

"calling activity" : The running activity, in whose program the executed subroutine call statement is contained.

"calling program" : The program which contains the executed subroutine call statement. See "calling activity".

"object activity" : The activity that is wanted or expected to be started, stopped, or otherwise affected as a consequence of a system subroutine call. It may also be termed "the designated" activity.

"object program" : Since all subroutine calls described here relate to object activities, rather than programs, the term "object program" will seldom be used. Where applicable, however, this term means the program, or a program, used by an activity under its execution.

Common parameter designations:

i specifies the object activity. Corresponding to previous working papers, this argument has been defined as either:

(a) an integer expression

or (b) a string of literals, designating an integer array name

or (c) a string of literals, designating an identifier name, like procedure name.

The processor shall define which of these forms are acceptable.

It might be discussed, however, if not this list advantageously could be made more restricted. The new Draft Proposed ANSI, FORTRAN Standard X3J3/76 ("FORTREV") contains a clear definition of "Names" and their scope. Even though "activity" is not mentioned in X3J3/76, FORTRAN's definition of names fits very well, for activities as well as for Programs, Subroutines, etc. Thus, the author feels it appropriate now to take advantage of the definitions of X3J3/76, and simply require the argument "i" to be:

"The name of the object activity".

Accordingly, I have simplified the text of the subsequent paragraphs, as compared to the previous edition of this document.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accepted
2 or greater : request rejected

This argument shall be an integer variable or integer array element, local to the calling program.

Topic for discussion: The numerical definitions above are according to all previous working documents. Why can't the values be the following, however?:

less than 0 : undefined
0 : request accepted
1 or greater : request rejected.

The author feels that the last list conforms more to common testing practice: In assembly languages, for example, the testing on zero or not is by far the simplest, zero indicating a "normal" situation. Even though most FORTRAN compilers might not take advantage of the ease of zero testing directly, most programmers are perhaps used to consider value zero to represent normal return. I have not changed the proposed text in the subsequent paragraphs however; I will await the outcome of a discussion.

The list of subroutine calls, described in detail in subsequent paragraphs is:

Full description in paragraph:	call:	parameters:
§ 6.2	CALL CREATE (h,i,j,m)	h : input name of created activity i : return designation of created activity j : descriptor of associated program
§ 6.3	CALL KILL (i,j,m) (opposite of CREATE)	j : termination parameters
§ 6.4	CALL START (i,j,k,m) (relative time)	j : numerical value of time delay k : time unit
§ 6.5	CALL TRNON (i,j,m) (absolute time)	j : reference to absolute time descriptor for activation instant

§ 6.6.1	CALL CYCLE (i,j,k,l,m) (cyclic, with delayed first activation)	j : length of time interval k : time unit l : time delay before first activation
§ 6.6.2	CALL TCYCLE (i,j,k,l,m) (cyclic, with absolute time spec. of first activation)	i,j,k,m: identical with param.of CYCLE l : reference to absolute time descriptor for activation instant
§ 6.7	CALL CON (i,j,k) (connection to event)	j : reference to event descriptor.
§ 6.8	CALL DECYCLE (i,m) (termination of periodic re-activation)	
§ 6.9	CALL DECON (i,j,m) (eliminate event connection)	j : like CON
§ 6.10	CALL CANCL (i,m) (eliminate scheduling)	
§ 6.11	CALL WAIT (j,k,m) (delay continuation of calling activity)	j : length of time delay k : time unit
§ 6.12.1	CALL WTS (s,m) (wait on semaphore)	s : semaphore variable (shared integer)
§ 6.12.2	CALL SIGNAL (s,m) (release semaphore)	s: semaphore variable
§ 6.12.3	CALL REGION (v,p,j,m) (entry of conditional critical region)	v : critical region designator p : priority j : event, condition for entrance to the critical region
§ 6.12.4	CALL REGEND (v,m) (textual terminator of critical region in program text).	v : critical region designator
§ 6.12.5	CALL AWAIT (v,j) (wait for event)	v : critical region designator, or zero. j : event specification for resumed execution
§ 6.13	CALL EXIT (termination of execution)	

6.2. Creation of a new activity.

A new activity is introduced to the real-time system by reference to subroutine CREATE. The designated activity will be associated with some specified program, considered a resource like other resources, necessary for the activity to perform. The associated program will normally reside on some background storage unit, administered by a file system. It may also be read in, for example from paper tape in simple systems, fetched from some separate place in the primary memory, etc., depending on parameter j of the call. Subject to restrictions by the processor, new activities may be created recursively. Thus, the same name (parameter h) may be specified as another activity, already in existence. For the purpose of being able to distinguish between similar activities, the executive system will supply, as return parameter i, a unique identifying name. This name, rather than parameter h, should be used as identifier in all succeeding system calls for state transitions of the created activity. If the processor denies recursive creation of activities, it will be natural to expect the executive system to return a name i identical to the input name h.

The form of the call is:

```
CALL CREATE (h, i, j, m)
```

where

h specifies an input name of the created object activity. If this parameter is identical to the actual name of an already existing activity, the executive system shall interprete this as a requirement for a new activity of the same type as the former. The argument is either the name of the object activity or an array name.

i is set on return to the calling program, as a unique designating name or identifier for the created activity, supplied by the executive system. This identifier is to be used in future reference of the designated activity, as long as it remains existent in the real-time system. The argument will be of the same form as parameter h.

j specifies an integer array which contains all information necessary to specify an associated program: its designation, where the program can be found such as description of file, residency while existent (core resident or swappable) etc. The details are processor-defined.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined.
1 : request accepted.
2 or greater : request rejected.

This argument shall be an integer variable or integer array element, local to the call.

6.3. Eliminating an activity from the real-time system.

A reference to subroutine KILL will eliminate a designated activity from the real-time system, by transferring it to state NON-EXISTENT. If the designated activity is in state DORMANT or PENDING, the effect shall be carried out immediately. If the designated activity is in any executing state, the termination shall only affect future executions. Thus, the designated activity will conclude its present execution, without any intervention by this call.

The processor may impose certain restrictions on which calling activities may be permitted to eliminate other activities. For example, a reference to subroutine KILL may be effective only if the designated activity has been created by the same calling activity, i.e. there exists a parent - child relationship.

The form of the call is:

```
CALL KILL(i, j, m)
```

where

i specifies the activity to be affected (object activity). The argument is either the name of the object activity or an array name.

j specifies an integer array which contains information to guide the termination, such as whether the object program is to be written to some background storage, the location or file name within such storage, etc. The specific details are processor-defined.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accepted. The designated activity (i) is terminated.

2	:	request accepted, but the activity is still executing within a run already started prior to the call.
3 or greater	:	request rejected.

This argument shall be an integer variable or integer array element.

6.4 Starting an Activity Immediately or after a Specified Time Delay.

Execution of a reference to the subroutine START shall, after the expiration of the specified time delay, cause the execution of the designated activity's first program unit. This execution shall commence at the program's first executable statement. The time delay is defined as the nominal duration from the time the call is made until the time the designated activity is supposed to enter state RUNNING. In the meantime the designated activity is listed in a table, and is transferred to state PENDING if the present state is, or when it becomes, DORMANT. The actual time instants for the entering and the leaving of the state PENDING are subject to the resolution of the system's real time clock, to the interrogating and activating actions performed by the Executive program, and to the possibility that the present state is not DORMANT. Thus, a reference to subroutine START is not necessarily equivalent to a transfer to state PENDING, but to a request of such a transfer, when it becomes permissible. A specified zero or negative delay shall be interpreted as "Immediate execution", which shall cause any residency in state PENDING to vanish or be as short as possible. The form of the call is:

```
CALL START(i, j, k, m)
```

where

i	specifies the activity to be executed. The argument is either the name of the object activity or an array name.
j	specifies the time delay, in units as specified by k, and as defined above. This argument shall be an integer expression.
k	specifies the units of time as follows: 0 - Basic counts of the system's real time clock 1 - Milliseconds 2 - Seconds 3 - Minutes 4 - Hours This argument shall be an integer expression.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accepted.
2 or greater : request rejected.

This argument shall be an integer variable or integer array element.

6. 5 Starting an Activity at a Specified Absolute Time.

Execution of a reference to the subroutine TRNON shall cause the designated activity's first program unit to be executed at a specified time (absolute time). Execution of the object program will commence at the program's first executable statement. The specified time shall be the time when the object activity is supposed to enter state RUNNING, and the actual time of this occurrence is subject to the resolution of the system's real time clock and to the interrogating and activating actions performed by the Executive program. The relations to pending-table and to transitions between states DORMANT, PENDING, and RUNNING are similar to those described for subroutine START. The form of this call is:

```
CALL TRNON(i, j, m)
```

where:

i specifies the activity to be executed (object activity). The argument is either the name of the object activity or an array name.

j Designates an array, whose first 6 elements contain the absolute time at which the object program is to be executed. These elements are as follows:

First element : Seconds (0 to 59)
Second " : Minutes (0 to 59)
Third " : Hours (0 to 23)
Fourth " : Day (0 to 31)
Fifth " : Month (0 to 12)
Sixth " : Year (0 →)

Any negative value is interpreted as zero, and any value greater than the upper limits specified above shall be taken as equal to the limit. Value zero for "Day", "Month" or "Year" shall be interpreted as current day/ month and/or year and should be changed to correct values automatically by the executive system.

This argument shall be an integer array name.
Time is interpreted as "current local time".

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accepted.
2 or greater : request rejected.

This argument shall be an integer variable or integer array element.

6.6 Starting an Activity in Periodic Execution.

6.6.1 Initial start immediately or after a specified time delay.

Execution of a reference to the subroutine CYCLE has the same immediate effect as a reference to subroutine START. Additionally, the designated activity shall be re-scheduled[1], each time the activity leaves state PENDING. The new time scheduled shall be equal to the sum of previous scheduled time and the interval specified by the call of CYCLE. The re-scheduling under said condition will continue until actively terminated by a call of subroutine DECYCLE, thus stopping a series of periodic executions. After such a termination, a new cyclic execution of the designated activity will require new call of CYCLE. The interval can be modified by another call of CYCLE, without an intervening call of DECYCLE. This modification shall, however, affect only succeeding scheduling operation, i.e. it shall not change the current interval and the execution which is already scheduled; the "delay" parameter of a modifying CYCLE call is without effect. If the "delay" parameter shall have effect, the previous cycling must first be terminated by a reference to subroutine DECYCLE. In this case, it will usually also be appropriate to eliminate the previous scheduling completely, by a call of subroutine CANCL.

---

[1] "scheduled" (written with small letters) is here in the meaning of "listed in the pending-table", as opposed to state PENDING. The distinction is explained in section 3.

The actual running may be delayed unintentionally, while in state RUNNING, because of running of other programs. Such delays will not be accumulated. If the execution is not finished before the time for next execution, the next execution will be delayed and started (re-scheduled) as soon as the previous execution is finished 2).

The form of the call is:

```
CALL CYCLE(i, j, k, l, m)
```

where:

i specifies the activity to be executed. The argument is either the name of the object activity or an array name.
Object and calling activity may be the same, i.e. argument "i" may designate the calling activity.

j specifies nominal length of the time interval, in units as specified by k , and as defined above. This argument sjall be an integer expression.

---

2) It should be discussed, how far such skewing effects may go before some alarm is generated. It should also be discussed, how an alarm can be implemented. One possibility, quite elegant with the present handling of programmed events, would be to affect parameter m of the cycle-call. This could, in turn, be used as a triggering event for a fault-handling program, initiated by CON. The effect would be that a sudden change of m to,say, 3, would start the fault-handling program.
Example:

```
CALL CYCLE (INTERGRATE, 2,1,F1)
----
CALL CON(FAULT1,FA1,m)
----
```

A suitable description in array FA1 could be:

=	F1	3

If this mechanism is allowed, the description of parameter m (5th parameter) of the cycle-call should be modified accordingly.

k	specifies the units of time as follows: 0 - Basic counts of the system's real time clock 1 - Milliseconds 2 - Seconds 3 - Minutes 4 - Hours This argument shall be an integer expression.
l	specifies a time delay, in units as specified by k, for the initial activation, as measured from the time the call was made. The time delay shall be interpreted like parameter j in call of subroutine START. This argument shall be an integer expression.
m	is set on return to the calling program, to indicate the disposition of the request as follows: 0 or less : undefined 1 : request accepted. 2 or greater : request rejected. This argument shall be an integer variable or integer array element.

6.6.2 Initial start at specified absolute time.

Execution of a reference to the subroutine TCYCLE has the same immediate effect as a reference to subroutine TRNON. Additionally, the designated activity shall be re-scheduled [1], each time the activity leaves state PENDING. The cyclic rescheduling is exactly identical to this function of subroutine CYCLE, described in section 6.5.1. The interval can be modified by another call to TCYCLE, or to CYCLE, with identical results as described in sec. 6.5.1. The form of the call is:

```
CALL TCYCLE(i, j, k, l, m)
```

where:

i, j, k and m are identical with the corresponding parameters of call to subroutine CYCLE (see sec.6.6.1.). l designates an array, whose first 6 elements contain the absolute time at which the specified activity is supposed to enter state RUNNING. This argument is exactly equivalent to parameter j of call for subroutine TRNON. The elements of the array are as follows:

---

[1] See footnote page 26

First element	:	Seconds	(0 to 59)
Second "	:	Minutes	(0 to 59)
Third "	:	Hours	(0 to 23)
Forth "	:	Day	(0 to 31)
Fifth "	:	Month	(0 to 12)
Sixth "	:	Year	(0 →)

Any negative value is interpreted as zero, and any value greater than the upper limits specified, shall be taken as equal to the limit. Value zero for "Day", "Month" or "Year" shall be interpreted as current day, month and/or year and should be changed to correct values automatically by the executive system. This argument shall be an integer array name. Time is interpreted as "current local time".

6.7. Connection of a program to an event.

Execution of a reference to subroutine CON shall cause the designated activity to be associated with a specified event and listed with a new entry in the schedule-table. This listing shall cause the transition to state PENDING if, or when, the designated activity is or arrives in state DORMANT. The specified event shall constitute the condition for the subsequent transfer to state RUNNING. The form of this call is:

```
CALL CON(i, j, m)
```

where:

i specifies the activity to be affected. The argument is either the name of the object activity or an array name.

j specifies en integer array which contains all the information necessary to designate the event to be connected and the type of connection.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accepted.
2 or greater : request rejected.

This argument shall be an integer variable or integer array element.

6.8. Terminating periodic execution.

Execution of a reference to the subroutine DECYCLE shall terminate any periodic re-sheduling of a designated activity, previously initiated by a call of subroutine CYCLE or TCYCLE. If the designated activity is PENDING or in any executing state, i.e. RUNNING or SUSPENDED, the termination shall affect only future executions. Thus, the designated activity will conclude its present and pending execution, without any intervention by this call. The form of the call is:

```
CALL DECYCLE (i,m)
```

where:

i specifies the activity to be affected. The argument is either the name of the object activity or an array name.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accespted. The object activity is not running.
2 : request accepted, but the activity is still executing within a run already started prior to the call.
3 or greater : request rejected.

This argument shall be an integer variable or integer array element.

6.9. Elimination of an event connection.

Execution of a call to subroutine DECON shall cancel any connection between a designated activity and a specified event. Thus, it eliminates further effects from a previous call to subroutine CON. If the object activity is in any executing state, the termination shall affect only future executions. Thus, the object activity will conclude its present execution, without any intervention by this call. The form of the call is:

```
CALL DECON(i, j, m)
```

where:

i specifies the activity to be affected. The argument is either the name of the object activity or an array name.

j specifies an integer array which contains all the information necessary to designate the event which is to be deconnected.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less	undefined
1	request accepted. The object activity is not running.
2	request accepted, but the activity is still executing within a run already started prior to the call.
3 or greater	request rejected.

This argument shall be an integer variable or integer array element.

6.10. Elimination of previous scheduling.

Execution of a reference to subroutine CANCL shall cancel all effects of previous calls to subroutines START, TRNON, CYCLE, or TCYCLE for a designated, object activity.

If the object activity is in any executing state (RUNNING og SUSPENDED), the elimination shall affect only future executions. Thus, the designated program will conclude its present execution, without any intervention by this call.

The form of the call is:

```
CALL CANCL (i,m)
```

where:

i specifies the activity to be affected (object). The argument is either the name of the object activity or an array name.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accepted. The object activity is not running.
2 : request accepted, but the activity is still executing within a run already started prior to the call or has still more entries in the pending-table, as consequences of previous calls of subroutine CON.
3 or greater : request rejected.

This argument shall be an integer variable or integer array element.

6.11. Delaying Continuation of an Activity.

Execution of a reference to the subroutine WAIT shall provide a means whereby a running activity is suspended for a specified length of time. After the suspending period, the program shall resume execution, first executing the instruction immediately following the call of subroutine WAIT.

The time delay is defined as the nominal duration from the time instant when the call was made until the program resumes execution in its virtual processor, by being transferred to state RUNNING. The actual time instants for the entering and leaving states RUNNING and SUSPENDED are subject to the resolution of the system's real time clock and to the interrogating and activating actions performed by the Executive system. A specified zero or negative delay shall be equivalent to no action, i.e. immediate return from subroutine WAIT. The form of the call is:

```
CALL WAIT(j, k, m)
```

where:

j specifies the length of time delay, in units as specified by k, and as defined above. This argument shall be an integer expression.

k specifies the units of time as follows:

0 - Basic counts of the system's real time clock
1 - Milliseconds
2 - Seconds
3 - Minutes
4 - Hours

This argument shall be an integer expression.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accepted.
2 or greater : request rejected.

This argument shall be an integer or integer array element.

6.12. Synchronizing suspension of an activity.

Syncronization between two parallel activities is achieved by transfer of one activity into state SUSPENDED while waiting for certain conditions to be satisfied by the other.

A number of different mechanisms for synchronization are proposed in the literature; each one with its attributes and claimed advantages. ([4],[7], to [13]). Here is proposed, to include in this standard of FORTRAN subroutine calls, two widely used mechanisms: Semaphores and Conditional critical regions. Each of these mechanisms has its class of problems, for which it is optimal in comparision with the other. It is generally acknowledged, that all synchronization problems may be solved by proper use of any one of them, and it is reasonable to assume, that these two mechanisms will be sufficient, also from a practical point of view.

Semaphores are traditionally manipulated by Dijkstra's P and V primitives. Although these primitives are widely known by these terms, single letter identifiers should be avoided, since they can be confused with user defined names. Thus, the semaphore manipulating subroutine calls are here described, using the suggested designations WTS (WaiT on Semaphore), and SIGNAL, corresponding to P and V respectively.

Conditional critical regions as means of synchronization are described by P. Brinch Hansen in [4] and [10] and convincingly advocated as particularly well suited for problems where several activities compete for exclusive access to shared resources. The author of the present paper has developed the principle further ([13]), by attaching priority to a call for entry into a critical region. In this paper, conditional critical regions with priority are managed by two system subroutine calls, REGION and REGEND.

Although theoretically superfluous because of the other synchronizing subroutine calls, it is useful from a practical aspect to extend the list of synchronization calls to include AWAIT:"Wait on event". Any event can be specified, and upon occurrence, the designated activity is transferred from state SUSPENDED and back to RUNNING.

The following paragraphs describe in detail the five subroutine calls for the synchronizing mechanisms mentioned in the introduction above.

6.12.1. Wait on Semaphore.

Execution of a reference to subroutine WTS involves manipulation on a non-negative integer variable, referred to by one of the arguments of the call, and shared with other, parallel activities. This particular shared variable is termed a "semaphore", and the immediate effect of the subroutine call depends on the magnitude of the semaphore, as decribed below. The operation on the semaphore shall be exclusive, i.e. only one activity at a time may access the semaphore *)

The form of the call of WTS is:

```
CALL WTS (s,m)
```

where:

s specifies an integer variable, termed the "semaphore" and shared with the parallel activities with which the calling activity is to be synchronized. The subroutine shall be granted exclusive access to this variable during its operation. If s is zero when the subroutine is called, the calling activity will be transferred into state SUSPENDED and the exclusive access rights will be released. An activity thus suspended will resume state RUNNING when the semaphore becomes positive again. At this point, subroutine WTS will perform, on behalf of the calling activity, the operation, exclusive in time:

$$s = s - 1$$

and a return is made to the calling program. If the calling activity is not suspended because s>0, the operation s=s-1 will be performed directly, without any intervening suspension.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accepted, synchronization obtained.
2 or greater: request rejected, synchronization is not performing as expected.

This argument shall be an integer variable or integer array element.

---

*) A somewhat less restrictive rule may be imposed, in fact: Simultanous reading may be allowed. This is of no concern in the present context, however.

6.12.2 Release of Semaphore.

Execution of a reference to the subroutine SIGNAL shall increment an integer variable, referred to by one of the arguments of the call, and shared with other activities. If this incrementation causes the numerical value to change from zero to positive, this event shall cause the transtition to state RUNNING for any other activity, waiting in state SUSPENDED for the occurrence of this event, relating to the same variable. The form of this call is

```
CALL SIGNAL (s,m)
```

where:

s specifies an integer variable, shared with the parallel activities with which the calling activity is to be synchronized. The subroutine shall be granted exclusive access to this variable during its operation and will execute the following modification of its value:

$$s = s + 1$$

A change from zero to positive value, caused by this operation, shall provide a releasing transfer to state RUNNING for any parallel activity waiting in state SUSPENDED for this event to occur.

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less	:	undefined
1	:	requested, synchronization obtained.
2 or greater	:	request rejected, synchronization is not performing as expected.

This argument shall be an integer variable or integer array element.

6.12.3. Entry of Conditional Critical region.

Execution of a reference to the subroutine REGION is a request from an activity to be granted access to its critical region, named as per argument v. The request is guided by a priority number supplied as argument p. The effect of the call is that the activity is transferred to state SUSPENDED where it shall remain until access to the critical region is granted. At this time, it will be transferred to state RUNNING. One condition for access to the critical region will be an event, specified as argument j. The release for transfer to state RUNNING shall not be performed until all the following conditions are satisfied:

1) v is not assigned to another activity
2) The event, specified by j, is true.
3) If more than one of concurrently executing parallel activities are simultaneously calling REGION, with the same parameter v, that one having the highest value of p, among those activities where all these conditions are satisfied, shall be selected for entrance into the critical region. This selection is indicated by *assigning* v to this activity, thereby preventing point 1 to be satisfied for the other activities. If the priority p has the same value for several activities, these activities will be selected in the order of their arrival to the queue.

The satisfaction of these conditions constitutes the *event* upon which the activity can be transferred from state SUSPENDED to RUNNING. At this point, the activity will have entered its critical region.
The form of this call is:

```
CALL REGION(v, p, j, m)
```

where:

v specifies an integer variable, shared with the concurrent programs to be synchronized. This parameter designates the critical region.

p specifies a priority and can be either
  a) an integer expression or variable
  or b) an integer constant
  in any case, the value of p shall be defined, prior to the call.

j specifies the *event* which constitutes a condition for the entrance to the critical region, as listed below. The argument is either:
  a) a real constant: value ≥ 0 implies the *event* is true; this is normally used for unconditional entry.
  or b) a real expression: value ≥ 0 constitutes the *event*.
  or c) an integer array name; the array contains all the information necessary to specify the event.

The processor shall define which of these forms are acceptable, and the processor also defines the specific form of array components, if applicable.

AD-A036 453 PURDUE UNIV LAFAYETTE IND PURDUE LAB FOR APPLIED IND--ETC F/G 9/2
SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIO--ETC(U)
JAN 77 N00014-76-C-0732

UNCLASSIFIED NL

2 OF 4
ADA036453

m is set on return to the calling program, to indicate the disposition of the request as follows:

0 or less : undefined
1 : request accepted
2 or greater : request rejected.

This argument shall be an integer variable or integer array element.

6.12.4 Exit from Critical region.

When an activity has entered a critical region, and a common variable v is assigned to it, this region can only be released, and v deassigned, by the same activity, by call of subroutine REGEND (or as a secondary effect of a call of AWAIT, see next section). The region, and the assignment of v, are considered similar to internal resources; thus the activity will formally remain in state RUNNING during and immediately after this call. The form of the call is:

```
CALL REGEND(v, m)
```

where:

v is the same variable as in a previous call of REGION.

m is set on return to the calling activity, to indicate the dispostion of the request as follows:

0 or less : undefined
1 : request accepted.
2 or greater : request rejected.

This argument shall be an integer varable or integer array element.

When the subroutine REGEND is entered, v will immediately be deassigned from the calling activity thereby exiting the calling activity from the critical region, and permitting v to be assigned to another activity, pending in its call of REGION.

6.12 5. Waiting for an event.

The waiting effect of synchronization is obtained through call of subroutine AWAIT, with the following effect:

The activity is suspended, until an event is true: The parameter j has similar effect as in call of REGION; it designates the event which subsequently will permit the calling activity to continue by being transferred to state RUNNING. The call will have no effect, if the specified event is true already when the call is made. The form of this call is:

```
CALL AWAIT(v, j)
```

where:

v specifies an integer variable, shared with parallel activities to be synchronized or, it can be the integer 0. This parameter designates a critical region and can be set equal to 0 if the call is not being used in connection with a critical region.

j specifies the *event* which constitutes the condition for the calling activity to resume execution by being transferred to state RUNNING. The argument is either:

a) an integer expression: value ≥ 0 constitutes the event.
or b) an integer array name; the array contains all information necessary to specify the event.

The processor shall define which of these forms are acceptable, and the processor also defines the specific form of array components, if applicable.

If the activity is within its critical region, designated by v, it shall leave this critical region immediately and permanently when AWAIT is entered, and v shall be deassigned. Thus, subroutine AWAIT includes the effect of REGEND. This ensures that concurrent activities, competing for the same critical region, are not unduly delayed if the programmer, inadvertently, puts the call of AWAIT inside the program section corresponding to the critical region. The only purpose of specifying the critical region, by v, is to achieve this automatic leaving of the critical region. Thus, it can be omitted or, the effect specifically suppressed, by using the integer 0. It would, however, be a good practice to always include the region specification, if applicable.

6.12.6. An alternative definition of critical region call.

If the call of subroutine REGION is extended with a fifth parameter, it is possible to eliminale the use of REGEND to terminate the scope of a critical region. The subroutine REGEND then becomes superfluous and can be delected completely from the set of standard subroutines. The new definition of the form of the call of REGION could be:

```
CALL REGION (v, p, j, l, m)
```

where v, p, j and m are as defined in section 6.12.3.

l specifies a statement label, indicating the last statement of the critical region. l is an integer variable name.

The statement label, indicating the end of the critical region,must be assigned to parameter l by a statement label assignment statement,i.e. a statement of the form

ASSIGN <u>s</u> TO <u>l</u>

where <u>s</u> is the statement label indicating the end of the critical region. For the assignment statement, all normal rules from ANS FORTRAN applies.

It should be discussed, which of the forms of REGION should be included in the standard: The pair REGION - REGEND as described in sections 6.12.3 and 6.12.4, or the alternative form of REGION as described in the present section and which makes REGEND superfluous. Both methods give the same number of statements in a program, and about the same number of rules to remember for the programmer. The method with assignment of a label appears to give slightly shorter execution time, since it eliminates one run-time call of a subroutine. This saving is probably only fictiticus, however: At the termination of the critical region, the executive system must be interrogated in any way, to release the region v for the benefit of other parallel activities. Perhaps this method even gives slightly <u>increased</u> execution time, because the executive system must be furnished with some mechanism to keep track of the execution of the region, to terminate the region when the last statement is reached. In any way,the realization, with regards to compiler and executive system, will probably be more complicated with the method in the present section. Since the advantage for the user is probably also questionable I see no reason at present time of the evaluation work to propose this method. It would be interesting to see other opinions, however.

6.13. Normal termination of execution.

Execution of a call to subroutine EXIT shall terminate the normal execution of an activity and return the activity to state DORMANT. The form of the call is:

```
CALL EXIT
```

with no parameters.

6.14. Status information of Pending-queue.

Execution of a call to subroutine QPEND shall provide, on return, information about all elements of the pending-queue (-table), regarding the designated activity.

The purpose is to supply, to the calling activity, all pertinent information about the designated activity's entries in the pending-table, prior to the call of CANCL or DECYCLE. This will enable the calling activity to reestablish those elements of the table, that are eliminated unintentionally by CANCL or DECYCLE. The call itself has no effect on state transitions.

The form of this call is

```
CALL QPEND (i, j)
```

where

i specifies the object activity. The argument is either the name of the object activity or an array name.

j will on return to the calling activity contain references to all elements of the pending-table that concerns the object activity. These references shall give all necessary information for the occurrences of i in the pending-table, such as execution time, cycle interval, and event connection. The argument is an array name. The specific details are processor-defined.

REFERENCES.

[1] Pettersen, O.: A Working Document towards a PROPOSAL ON TASK MANAGEMENT. SINTEF, The Tech.Univ. of Norway/Stanford University, March 1975.

[2] Pettersen, O.: A NOTE ON SYNCHRONIZATION OF CONCURRENT PROCESSES. Background for proposal on FORTRAN subroutine calls. The Tech. Univ. cf Norway/Stanford University, Jan./April 1975.

[3] WORKING PAPER, "Minutes" 8. Workshop, October 1972, pp 63-86.

[4] Hansen, P.B.: Operating System Principles. Prentice-Hall 1973.

[5] Heller,G. et al.: Prozess-FORTRAN 75, eine Erweiterung von FORTRAN für Prozessrechner-Anwendungen. Vorschlag des Arbeitskreises "Prozess-FORTRAN" der VDI/VDE-Gesellschaft für Mess- und Regelungstechnik. (Dec. 1975).

[6] Revised S61.1, Minutes from 1974 Spring Regional meetings of Workshop, pp 42-63.

[7] Pettersen, O.: A Note on Synchronization of CONCURRENT Processes. Working document for the Fortran Committee,Purdue Workshop, January 1975.

[8] Dijkstra, E.W.: Cooperating Sequential Processes. In Programming Languages (V. Genuys, ed.), American Press, N.Y.1968,pp43-112.

[9] Hoare, C.A.R.: Towards a theory of parallel programming. International Seminar on Operating Systems Techniques. Belfast, Northern Ireland, Aug.-Sept. 1971.

[10] Hansen, P.B.: A Comparison of Two Synchronizing Concepts. Acta Informatica 1(1972) pp. 190-199.

[11] Habermann, A.N.: Synchronization of Communicating Processes. Comm. ACM 15, 3(March 1972) pp. 171-176.

[12] Hoare, C.A.R.: Monitors: An Operating System Structuring Concept. Comm. ACM 17,10 (Oct.1974) pp. 549-557.

[13] Pettersen, O.: Synchronization of Concurrent Processes. Submitted for publication. (February 1975).

[14] Shaw, Alan C: The logical Design of Operating Systems. Prentice-Hall, 1974.

[15] Hoare, C.A.R.: Hints on the design of a Programming Language for Real-Time Command and Control. In Course text "REAL-TIME SOFTWARE". Infotech'76.

[16] Hansen, P. B.: The Programming Language Concurrent Pascal. IEEE Trans. on Software Engineering SE-1, 2 (June'75) pp. 199-207.

[17] Knuth, D. E.: Structured Programming with go to Statements. ACM Computing Surveys. Vol. 6, No. 4 (Dec.1974) pp. 261-301.

[18] Pettersen, O.: MANAGEMENT OF PARALLEL ACTIVITIES IN REAL-TIME FORTRAN. State Model and State Transitions. Report No. 76-82-W from The Norwegian Institute of Technology, Div. of Engineering Cybernetics. 1. edition, March 1976.

SECTION IV

A REAL-TIME FILE SYSTEM AND JOB INITIALIZATION ROUTINES FOR PROCESS CONTROL COMPUTERS

The material contained herein was contributed to the Workshop through the courtesy of Royal Dutch Shell Petroleum, The Netherlands. It was presented at the Fourth Workshop on Standardization of Industrial Computer Languages, November 1970, by Mr. Antonius M. Hens, and published in the Minutes of that Workshop, pp. 234-277. This material was very helpful to the Workshop participants in their discussions of file handling procedures.

A REAL TIME FILE SYSTEM AND JOB INITIALIZATION ROUTINES FOR PROCESS CONTROL COMPUTERS

Contents

1. Real time file system
2. Job starting and completion routines
3. Auxiliary routines
4. Figures.

N.B. THE FILE SYSTEM ROUTINES ARE AVAILABLE FOR
CONTROL DATA 1700 CONTROL DATA NEDERLAND NV RYSWYK/HOLLAND
(IBM 1800 IBM BRANCH OFFICE MANCHESTER U.K.)
THE other ROUTINES may also be obtained through
control data nederland in Holland. They are not available
on IBM 1800.

This document is confidential. The copyright therein is vested in the Bataafse Internationale Petroleum Maatschappij N.V. The Hague, the Netherlands. Recipients must obtain the written consent of the said company before wholly or partly duplicating the contents or disclosing same to others. All rights reserved.

PRECEDING PAGE BLANK-NOT FILMED

1. REAL TIME FILE SYSTEM

1.1 Definitions (Fig. 1)

1.1.1 A file consists of a number of blocks (NBLK) and each block of the file contains an equal number of words (LBLK = block length). This length is to be equal to or an integral fraction of the "record length" of the logical unit on which the file resides (i.e. 1, 2, 3, 4, 6, 8, 12, etc. but not 5, 7, 9, etc if this record length is 96).*) if it is to be used in SECTOR ADDRESSABLE MODE ***)
The total number of words in a file is therefore equal to NBLK * LBLK**).

1.1.2 A file is specified as being a member of a set of files belonging to a certain unit. Therefore, NUNIT, NFILE specifies file NFILE of unit NUNIT. Unit numbers start with unit 0, files with number 1. Note that the files UNIT 0 File 1 to 15 are reserved for the file system itself.

1.1.3 A file is always processed (read or written) by block rather than by element. Consequently, to use a file a block pointer (IPNT) is required.

---

*) 96 is the number of words of a CDC 1700 disk sector. Some files will have a "record length" which is equal to LBLK which will for example be the case if the logical unit is a card reader, line printer, typewriter, etc, or when a special bit in the file definitions block is set.
In the case of a CDC 1700 computer this will mean "word addressable" mode.

**) An exception on this rule are files located on in- or output devices (line printer etc.). In such a case NBLK may also be undefined.
The record length for such files is always equal to the block length LBLK.

***) In word addressable mode any block length (<512) may be used

This document is confidential. The copyright therein is vested in the Bataafse Internationale Petroleum Maatschappij N.V. The Hague, the Netherlands. Recipients must obtain the written authority of the above company before wholly or partly reproducing the contents or [illegible]

1.1.4 Files may be resident on different logical units (i.e. core, disc, drum, CARD READER/PUNCH, LINEPRINTER, etc.).*)
Core working area to allow the use of files which are not core resident has to be specified by the user (IBUF see section 1.2.2).

1.1.5 Each file has a file definition block (FDB) which indicates to the system on which logical unit the file resides together with its "~~record length~~ mode" bit,***) its block size and number of blocks and whether the file is in a protected or unprotected status (see section 1.2 and Fig. 1).
An extra protect bit to prevent accidental and erroneous deletion of files (see section 1.2.1) is included.**)
File definition blocks of so-called rotate files have partly the same specifications as above. They are separately treated in section 1.1.7.

1.1.6 To use a file a block pointer (IPNT) has to be defined. The block pointer consists of 10 locations, and has to be defined by a dimension statement in the user's program, e.g. DIMENSION IPNT (10).

Each block pointer is unique in the sense that in a multiprogramming real time system the block pointer is only known to the program which has defined it.

---

*) But a particular file resides only on one particular logical unit.

**) Although it is no part of this specification, file definition block should preferably be so organized that:

a. they can be resident on different logical units (core drum, or disc) ~~[illegible]~~
b. they are partly core, partly drum or disc resident;
c. that spare file definition blocks can be reserved on the different logical units.

In selecting the possibilities in the file definition block organization it is essential that a high flexibility is not obtained at the expense of an excessive consumption of core space and/or execution time.

***) mode: sector or word addressable 1 or 0

AMENDMENT

to BICM-CMF/4 MEMORANDUM OF DISCUSSION No. 149/70),
page 3, point 9

bit 15	R/W controls adjustment of 1 and 2 *
bit 14	File protect bit ex FDB
bit 13	Mode (sector or word addressable) ex FDB
bit 12	Information has been read into core indicator
bit 11	Flag indicating that a write command is not yet executed.
bit 10	Flag indicating that the blockpointer is part of a background program (i.e executed in unprotected core)

The block pointer contains:

1. A self-relative (core) address of the block of the file to which this pointer is pointing; **)
2. the number of the block to which the block pointer is pointing;
3. The number of blocks of the file NBLK
4. The number of words of each block LBLK
5. The "file address"
6. A pointer to the file protect bit
7. The number of the first in core block
8. The number of blocks which can be in core (in the space set aside for the file in IBUF)
9. the file logical unit number, together with the following four control bits:

   bit 15 R/W indication to control the adjustment of 1 and 2*)

   bit 14 indicator whether information in core is originally been read in from the file attached to the block pointer

   bit 13 flag indicating that a write command is still not yet executed

   see Amendment

   bit 12 "Word addressable mode".
10. the address of the error exit in the user's programme (section 1.2.2). Upon errors it is left to the user to decide which actions he will take. If no address is specified the testing of an error will result in a fatal massage and stopping of the job.

---

*) The adjustment of the block pointer is dependent on the previous operations performed ("read"/ "write" and or "attach" operations).

**) This allows the communication in fortran with the contents of the file by stating EQUIVALENCE (IP, IPNT(1))

the contents of the first word of a block is found in

IPNT (IP)

second word IPNT(IP+1) etc, etc.

1.1.7 A number of files, each with the same number of blocks and of equal block length, may be arranged into a "rotating" or cyclic file (Fig. 2). The file definition block of a "rotating" file gives the system information about the starting address of the set of files, the number of files, the space (e.g. number of sectors) occupied by one file and a pointer to the current file in addition to the normal information (see section 1.1.5) contained in the definition blocks. The file definition block of a rotate file occupies therefore the double amount of space (6 words) as the regular file definition blocks and uses therefore two file numbers.
Two special routines are provided which attaches one of the files from such a "rotating" file to a block pointer.

The second half of the file definition block of a rotate file may be regarded as a regular file definition block of a so-called "slave" file. This "slave" file definition block has always the "current" file in its file address. The "slave" file may therefore be used as if it were a normal file (see section 1.2.9).

1.2 File handling routines

1.2.1 Call Define (NUNIT, NFILE, NBLK, LBLK, LU, IND) *)

Call Delete (NUNIT, NFILE, IND)

The Define and Delete routines will allow the dynamic generation or deletion of files:

NUNIT	unit number of the required file
NFILE	file number, if this file number is equal to zero on entry, the routine will return a file number to the user; if NFILE is not equal to zero, the specified file will be defined (see also IND).
NBLK	number of blocks of the file
LBLK	block length
LU	logical unit number e.g. 0 = core, 1 = drum, 2 = disc, etc. (add 32 for sector addressable mode) **)
IND	= 0 no error detected, file is defined. ERRORS ARE FOR EXAMPLE: attempt made to redefine an existing file no spare available on requested LU no file definition block available NFILE out of range NUNIT out of range

Note: ALL ERROR MESSAGES ARE ARRANGED IN A FILE OF ERROR MESSAGES. IF AN ERROR OCCURS IND CONTAINS THE BLOCK NUMBER OF THE MESSAGE IN THAT FILE

*) MAY ALSO BE USED AS FUNCTIONS. IN THAT CASE A REGISTER WILL CONTAIN THE VALUE OF IND. PROGRAMMING MAY BE DONE AS FOLLOWS: IF(DEFINE(.–, ––, ..., ..., .–, .–))ER,OK,ER ONLY IF THE OUTCOME IS ZERO GOTO OK

**) LU = 0 (and 32) will always mean core. FOR OTHER LOG.UNITS There is a translation table between file system logical units and CDC operating system logical units.

Note: ~~It is to be considered to place~~ An extra protect bit is placed in the file definition block in order to prevent accidental and erroneous deletion of files (e.g. fixed files belonging to a particular system. This protect bit is then to be set by a special initialization routine *)).

1.2.2 CALL ATTACH (NUNIT, NFILE, IPNT, IBUF, LENGTH, IERRXIT)

Attaches file NUNIT, NFILE, **) to block pointer IPNT. The block pointer is 10 locations and has to be defined in the user's program by a dimension statement as: DIMENSION IPNT (10).

IBUF	gives to the routine the address of file work area (for non-core files.). The dimension of IBUF should (again only for non-core files) be sufficient large to contain a complete disk sector, or a complete block (LBLK) in case of word addressable mode. It should be considered whether the use of an external IBUF may be advantageous.
LENGTH	specifies the number of words in IBUF
IERRXIT	gives the address in the user program to which control is transferred in the case of a read, write or other possible fatal errors. The address in the argument is set by an ASSIGN XXX TØIERRXIT statement. If the argument is not used (i.e. = 0) the job will come to a fatal end if an error occurs with a short error message. Further result will in such a case be unpredictable. ****)

Upon return from the attach routine IPNT(1) will point to the first block of the file***) IPNT(2) will contain a one (1) for block number one; IPNT(3) and (4) contain the number of blocks and the block length (NBLK and LBLK) of the file. See for the contents of the rest of IPNT the description given in section 1.1.6.

---

*) Such a routine can be made with the following call: CALL PRØTFL (NUNIT, NFILE), or by a special version of Define such a bit may be set. This solution is finally chosen.

**) It will advantageous to specify NFILE as external. In this way the file number can be assigned at system generation time without the need of recompilations etc.

***) more accurate IPNT(1) is pointing to the beginning of the core work area IBUF which is the potential location for the first block as soon as it will be made (or read in).

****) Note that the error exit address is saved in the blockpointer [illegible] to any of the other

1.2.3 CALL PRØTEC (IPNT)

CALL UNPRØT (IPNT)

These calls put the file to which IPNT has been attached (see 1.2.2) into a protected/unprotected status. This means:
when a file is placed in a protected status no other program than the one which issued the protect, is able to write into that file (see also 1.2.6 and its note).

Note: If a file is already protected when an attempt is made to protect that file, the program which requests this protect is suspended at the CALL PRØTEC statement until a CALL UNPRØT (from another program) clears the protected status of that file. Since this situation may occur with more than two programs using the same files, queuing of the suspended programs is required. (This will mean saving of all registers, the priority, etc., queuing according priority and within a priority on a first in first out basis).

The call UNPRØT routine has to check whether any program has been suspended because of a call PRØTEC for that particular file which is now to be unprotected. If there is such a program waiting on the queue it is to be restarted and taken of the queue*).

1.2.4 CALL SEQRD (IPNT)

The first time SEQRD "sequence read" is called after attaching the file to IPNT the first block of the file is read. At each subsequent call the next block is read in. Upon return IPNT is always pointing to the block just read. It is also possible to use SEQRD after a CALL RANDRD (section 1.2.5). Upon exit IPNT(1) is pointing to the block just read. IPNT(2) contains the block number of that block.

1.2.5 CALL RANDRD (IPNT, NBLØCK)

"Random read" will read the specified block NBLØCK and returns the block pointer IPNT pointing to the block NBLØCK of the file attached to IPNT. IPNT(2) will contain the value of NBLØCK at return from random read. As an alternative, the name RDBLK (IPNT, NBLØCK) "read block" may be used.

---

*) Note that in the queue which will be probably a threaded list, the cells of the list must not only contain the original registers and priority and suspension address, but also an indicator which uniquely indicates for which file the program has been suspended.

1.2.6 CALL SEQWR (IPNT,IRETURN)
CALL WRSTP (IPNT)
CALL SEQEND (IPNT)
CALL FINWR (IPNT)

After a file has been attached to IPNT, blocks of the file can be written with the aid of the above routines. The file is first to be placed in a protected status by a previous CALL PRØTEC (IPNT). If the file is not placed in a protect status the above routines will issue an error message and return to the error exit IERRXIT as defined in section 1.2.2*).

The first time SEQWR or WRSTP "sequence-write" or "write-step" is called after attaching the file to IPNT the first block of the file is, in principle, written**). At each subsequent call the next block is written. Upon return IPNT(1) is always pointing to the block directly after the one just written. IPNT(2) contains the block number of that block to which IPNT(1) is pointing.

In SEQWR the second argument IRETURN is an optional return address (set by an assign statement, e.g. ASSIGN 10 TØ IRETURN) to which control is given as long as the end of the file has not yet been reached. After writing the last block of the file control will be given to the next executable statement. This optional return offers the possibility of performing a so-called DØ-loop over an entire file.

---

*) Note: Although it is conceivable that another program which is not part of the job under consideration, did put the file in a protect status, it is highly unlikely that these two programs will always run in the same time slice especially because each program will have to pass a test phase during which the other program is probably not in operation. Hence, any forgotten CALL PRØTEC will already during testing be found because the fatal error message.

**) "in principle" because of efficiency reasons write commands are only then executed when a complete core buffer IBUF is fully used, thus making space for a next core load, or when file processing is finished by FINWR or SEQEND.

The CALL SEQEND*) will be used to end sequential file procedures (specially file write procedures SEQWR, WRSTP) in a correct way.

Note: that if the SEQWR routine is used, and the normal exit has been reached (end of the so-called DØ loop) a call to SEQEND will be regarded as a dummy operation and may therefore be omitted. This is of course also true for WRSTP, if the in core blocks of the file were truly written during the last time it was executed.

FINWR "final write" is a routine very similar to SEQEND but will always imply a write command. A call to FINWR has therefore the exact same effect as a call WRSTP directly followed by a call SEQEND.

Note: All the above routines can also be used after a previous call to RANDRD (IPNT, NBLOCK).

.2.7 CALL COPYFL (NUNIT 1, NFILE 1, NUNIT 2, NFILE 2, IERRXIT)

This routine copies one file into another**). Note that the operation of copying is done under protection of the files concerned, which again may result in job suspension and queuing as described for PRØTEC and UNPRØT (see section 1.2.3).

ERRXIT contains the address to which control will be given if a read/write error during copying occurs. It is set by an ASSIGN XXX TØIERRXIT statement.

Note: The user can perform his own copying operations by making a duplicate block pointer by using DUPIPT (see section 1.2.10)

---

*) The SEQEND routine is required because a file is read or written per block, however, if a file is for example disc resident, many blocks may go into a sector. The true read write operation to or from disc is, however, done sector-wise in order to economize on disc transfers. Therefore, this special routine which will transfer the correct number of words when writing is stopped somewhere halfway the file.

**) A minimum requirement is that the second file has at least the same number of blocks and has a block length equal to the block length of the first file.

1.2.8 CALL COPYCL (NUNIT 1, NFILE 1, NUNIT 2, NFILE 2, IERRXIT) **)

This routine copies file NUNIT 1, NFILE 1 into file NUNIT 2, NFILE 2 as with COPYFL but also clears file NUNIT 1, NFILE 1.
The operation is again performed under protection of the files as is described in 1.2.7.

Note: It may seem that with a separate clear file route **) (e.g. CLEARF) all needs will be fulfilled. The routine described here could then be made by:

```
CALL COPYFL (.......)
CALL CLEARF (.......)
```

but this will mean that between COPYFL and CLEARF the file will be unprotected (and may consequently be updated (and protected) by another program (of higher priority). To prevent this situation the calls have to be combined into a call COPYCL. Since this call has to be available no real need exists any more for a clear file routine.

1.2.9 CALL RØTATE (NUNIT, MFILE, IPNT, IBUF, LENGTH,IERRXIT) CALL HISTØR (NUNIT, MFILE, NDELAY, IPNT, IBUF, LENGTH IERRXIT)

These routines are provided to communicate with so-called "rotating" files (see section 1.1.7)*), and performs the same function as the ATTACH routine (section 1.2.2).

---

*) A special initializing routine (similar to DEFINE) is required to initiate the file definition blocks of the rotate and slave file. See also next note.

ALSO a clear file routine is available. It's call is

CALL CLFILE (NUNIT, NFILE)

and not clearf.

CALL RØTATE will attach the current file, to which the current file pointer is pointing, to the block pointer IPNT and increases the current file pointer to point to the next file in the rotating of cyclic files. At the same time the file address in the second part of the file definition block will be set to the address of this "next" file*).

CALL HISTØR attaches the file which is NDELAY files before the file to which the current file pointer is pointing. The current file pointer is, however, not changed. (If NDELAY = 0 is used the same file out of the rotating files is obtained as if CALL RØTATE were used, but without readjustment of the current file pointer).
MFILE is the name (number) of the rotating file.

1.2.10 CALL DUPIPT (IPNT, JPNT, NUNIT, NFILE)

makes a duplicate block pointer JPNT from block pointer IPNT (i.e. pointing to the same work area) but attached to file NUNIT, NFILE. Note that only the 5th, 6th and 9th location of the block pointer JPNT (see section 1.1.6) are obtained from the special file definition block NUNIT, NFILE. The rest is a "copy" of the block pointer IPNT, therefore by making a call ATTACH for NUNIT, NFILE and JPNT but using the same buffer IBUF and error exit ERRXIT does not necessary give the same result**).

1.2.11 CALL RESIPT (IPNT)

Reset IPNT to the beginning of the file attached to it (i.e. IPNT (1) points relative to itself, to beginning of work area IPNT (2) = 1).

1.3 Using the File Organization Routines

Considerable care should be taken to ensure correct use of the file routines. Any misuse may lead to unrecoverable errors and program stoppage.

---

*) As mentioned in section 1.1.7 the file definition block of a rotate file is equal in twice the number of words for a regular file definition block. Because of the same layout the second half may be used as the file definition block of a so-called "slave" file with file number equal to MFILE + 1. Therefore it is possible by following a call RØTATE by a call ATTACH for MFILE + 1 to obtain two block pointers (IPNT and JPNT) pointing to two files in the rotating files which are next to each other. (i.e. at completion of these routines the current - 1 and the current file).

**) This may be of importance if one wants for example print a text file from disk on the line printer. One makes now a duplicate block printer attached to the "line printer resident" file, but with the number of blocks NBLK and block length LBLK of the disc resident text file. By using now SEQRD and SEQWR with the thus obtained block pointer a complete report (file) may be printed out

Note that on a process control computer one can have several jobs (which may be not related to each other) where each job may consist of a number of programs (in Fortran PROGRAM statement) or overlays. Each program may use several subroutines. Consequently the words Job, Program and Subroutine will be used with the above meaning.

In the following some general rules are given:

1.3.1

In each program the bock pointers used should be dimensioned and equivalanced, e.g.:

```
DIMENSION      IPNT (10)
EQUIVALENCE  (IP, IPNT (1~~0~~))
```

Note that according to the definitions, IPNT (1) is relative to itself.

This means that for example

```
IPNT (IP)
```

gives (in Fortran) the contents of the first element of the block to which IPNT is pointing.

```
IPNT (IP + 1)
```

gives the contents of the second element, etc.

Very often it is possible to give selfexplaining names to the contents of a block such as high limit, low limit desired value and dead band for a block containing alarm limits. By using the following way of coding:

```
DIMENSION  IALM (10), IBUF (96)
EQUIVALENCE (HLIM, IAL, IALM   (1)), (LLIM,IALM (2))
EQUIVALENCE (DESVAL, IALM (3)), (DBAND, IALM (4))
INTEGER HLIM, DESVAL, DBAND
DIMENSION  HLIM (1), LLIM (1), DESVAL (1), DBAND (1)
```

in which IALM is the block pointer of the alarm limits file, the values of the alarm limits in the block (after reading a block of the alarm limit file by one of the read routinesdescribed in sections 1.2.4 and 1.2.5) can be found in the locations:

HLIM (IAL)	for the high limit
LLIM (IAL)	for the low limit
DESVAL (IAL)	for the desired value
DBAND (IAL)	for the dead band.

When floating point numbers are used in the files special care has to be taken. By using an integer variable equivalenced with a floating name problems are solved:

```
EQUIVALENCE (A, IA) (DIMENSION IA (2))
```

Then IA = IPNT (IP)
IA(2) = IPNT (IP + 1) gets the floating point figure from the file into A.

1.3.2 Use IP as argument in the file handling routines instead of IPNT (e.g. CALL SEQRD (IP)). This allows the fortran compiler to detect that the subscript IP of IPNT in a following statement with IPNT (IP) has or may have been changed. Fortran will therefore recalculate the index and will not assume an unaltered index IP.

1.3.3 Before using a file in a program first attach the file to a block pointer by the ATTACH routine.

1.4 Initialization of file definition blocks (preliminary) *)

A special mass storage resident program is needed which allows the initialization of the [illegible] file definition blocks.
This program can probably make use of the DEFINE routine. The program will read in file definition cards of the following format:

A2, 3X, 7I5.

The cards contain according to the above format following information:

a. for a normal file

FL, NUNIT, NFILE, NBLK, LBLK, LU, N0P , IEXTR **)

b. for a rotate file

RF, NUNIT, NFILE, NBLK, LBLK, LU, NOFILES, IEXTR

where

LU = logical unit of file (incl. mode ~~record length~~ bit) i.e. add 32 FOR SECTOR ADDRESSABLE)

NOFILES = number of files = 0 for non-rotate files.

Special control cards will allow the optimization of the location within a certain logical unit of the files for example:

NC start following file cards for the disc on a new disc cylinder.

The Format for this card may be A2, 3X, I5 and contains

NC, LU

where LU is again the logical unit.

*) A very general initialisation routine is now available (FSINT with a extensive number of options. It will also print out a listing of defined files etc, together with the relevant data (NBLOK, LBLK etc)

**) N0P is a dummy, IEXTR ALLOWS The reservation of some extra blocks (as a reserve) By a special routine it is then later possible to increase the number of blocks "on line" without the need to redefine all the files.

2. JOB STARTING AND COMPLETION ROUTINES

2.1 General[x]

Each program (separate overlay) of a job has an entry in the system directory. It is this entry which is threaded into the schedule stack when a schedule request is issued.

Because of the above a program cannot be rescheduled again until the first schedule has been honoured (program taken into execution).

If rescheduling is tried when the first schedule is still in the schedule stack the request is rejected and control returned with the Q register set negative.

One other important facet of rescheduling a program is that the program has to be re-entrant[xx]. However, this requirement cannot be met for most of the jobs. It is therefore of the utmost importance that a program or job is not rescheduled (taken into execution again) before the program has terminated a previous execution.

The above two facets are the reason for providing special (re-entrant) subroutines which are to be used to start and to terminate a job. An additional feature of these subroutines is that jobs which require disc transfers can be put into a delay status in order to provide for disc maintenance time (0-30 min.) without the loss of pertinent information.

2.2 CALL INITJB (NAME, IARG, IND)

CALL STRTJB (NAME, IARG, IND)

CALL ENDJØB (NAME, MAIN)

where:

NAME is the name of the job. Name is to be defined as external: EXTERNAL NAME

IARG is the name of a dimensioned array which contains the input arguments (if any), for the job. If no input arguments are required a zero (0) is to be used.

---

[x] Although it may seem that these remarks apply only to a CDC 1700 computer similar observations will apply to most process control computers.

[xx] At least when it is a core resident program.

IND is an error indicator
= 0 no error
= 1 no room available to store arguments
=.- negative request to start job rejected (fatal error).

MAIN is the name of the program which issues the call to ENDJOB. This name (MAIN) is used to release core of this program part (Note that a job may consist of more than one consecutive program).*)

INITJB and STRTJB are almost the same routines. The only difference is that that if INITJB does not accept the arguments because the table of arguments for that job is already full (IND = 1) STRTJB can still accept one set of arguments.

The arguments are stored in cells of a cylindrical list, the length of the list determines how many sets of arguments can be accomodated (the list has one spare cell which may be used by STRTJB).

2.3 Job tables

In order to be able to use the calls as described in 2.1 and to incorporate other features such as the automatic starting of jobs at a predetermined time interval etc. a special program which should be under the responsibility of the system supervisor of a particular installation is required and which can be called "Job tables" (JBTAB).
All the NAME arguments of the previous section (2.1) which are the names of all the real time jobs, should be entry points in this JBTAB program.

For each job in this job table program the following information is to be available:

1. The (initial) priority of the job.
2. Whether the job is to be restarted when a second etc., call is made for the job while it is still busy with a previous call.
3. Whether the job may be placed in a delay status for disc maintenance.
4. Whether a special job (so-called repeat job) is to be started when a secondary, third, etc., call is made for the job which is still in execution (see also under point 2).
5. Whether the job is to be started at a predetermined time interval by a "clock" routine. In this case the job must have a count-down location and an indication of the required time interval.

---

*) On CDC 1700 use is made of the call RELESE (MAIN) for the other program parts not ending with a call END-JOB. The call RELESE of the Fortran run time package is modified so that also core resident programs can issue a call RELESE without causing errors (Same rules as for release request)

6. For a job with arguments the arguments must be saved in a cylindrical list (per job) on a first in first out basis. A special routine is required for these jobs to obtain access to these arguments (see section 2.3).

2.4 CALL JØBARG (NAME, INHC)

Jobs started with one of the calls given in section 2.1 and which use arguments must have the means to get the value of these arguments from the cylindrical table (see section 2.2) in which they are stored.

For this purpose the JOBARG routine has been provided. The routine may be used repeatedly in the same program or set of programs of a job. An example is:

```
DIMENSION  INHC (1)
EQUIVALENCE (INH, INHC (1))
EXTERNAL NAME (name of the job)
CALL JØBARG (NAME, INH)
```

after execution of the above call

INHC (INH) contains the first argument (IARG (1))

INHC (INH + 1) contains the second argument etc.

3. AUXILIARY ROUTINES

3.1 CALL MASK
CALL DMASK

These are two special routines which, although used in the file organization routines themselves, do not belong to the file system proper.

The routines make it possible to use non-re-entrant non-inhibited Fortran callable routines in a multi-programming environment by first making a call to MASK and after the routine a call to DMASK. The routine or routines called between these two statements should, however, in no case make any I/O (input/output) command or request.

It is important that no call to one of the file organization routines is made if a CALL MASK is still effective.

3.2 CALL LØCAL (NAME, IBUF, IARG 1, ...IARGn)

This routine "loads on call" the subroutine NAME into work area IBUF.
The number of arguments (IARG) of the subprogram must match with the number of IARGs in the call LOCAL.
The effect for the user is therefore the same as if he used

CALL NAME (IARG 1, ...IARGn)

The work area IBUF should be at least equal to the length of the subprogram.

Note: NAME will most likely be a core resident array of two locations containing the address of the subprogram on disc and the program length. Therefore name has to be specified as external. A special program may put the address and length information in the array after the subprogram itself has been placed as a file on mass memory storage.

4: FIGURES

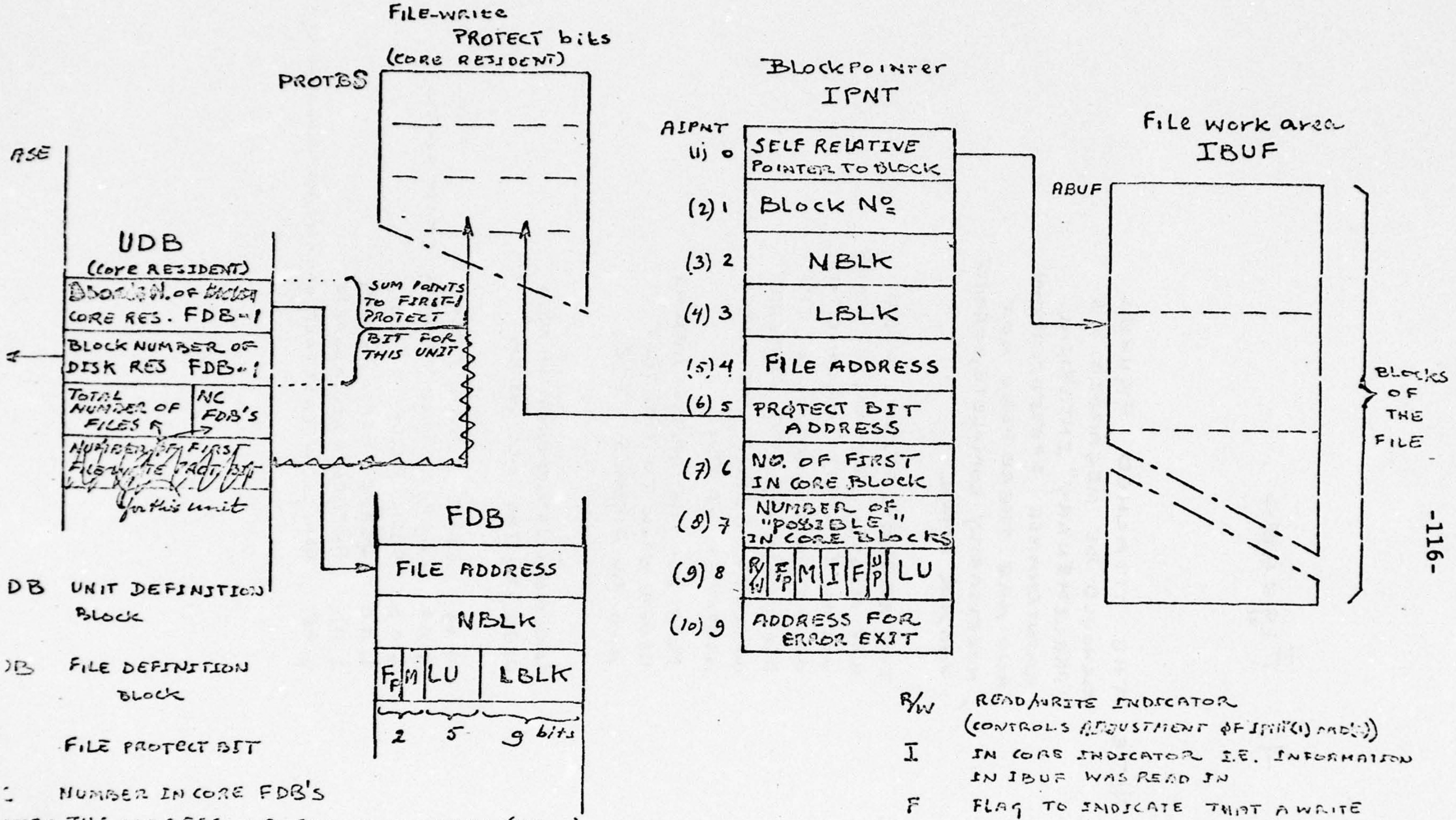
NOTE THE ATTACHED FIGURES SCHOULD BE REGARDED AS "PRELEMINARY" INTERNAL MAINTENANCE SPECIFICATION AND ARE THEREFORE NOT NECESSARILY COMPLETE, CORRECT AND/OR FINAL

THE ABBREVIATIONS USED ARE MOSTLY SELF EXPLICATORY WITH IPNT(1) THE CONTENTS OF THE FIRST LOCATION OF THE BLOCKPOINTER IPNT IS MEANT WHEREAS AIPNT THE CORE ADDRESS IS OF IPNT.

MOST OF THE ABBREVIATIONS REVER BACK TO THE TEXT AND TO FIGURES 1 & 2.

LOCATIONS IN COMMUNICATION AREA USED BY THE FILE SYSTEM

$A9 JSAVE SAVE I REG.
$AA CURPR SAVE PRIORITY (MASK ROUTINE)
$AB MASK PRIORITY (9)
$AC QSAVE SAVE Q REG
$AD RETURN BASE ADDRESS
$AE ADDRESS OF LAST PART OF READ/WRITE ROUTINE (Fig. 11 d.)


File-write
PROTECT bits
(CORE RESIDENT)
PROTBS
ASE
UDB
(CORE RESIDENT)
BLOCK N. OF FIRST
CORE RES. FDB-1
BLOCK NUMBER OF
DISK RES FDB-1
TOTAL NUMBER OF FILES
NC FDB'S
NUMBER OF FIRST FILE WRITE PROT. BIT
for this unit
SUM POINTS TO FIRST-1 PROTECT
BIT FOR THIS UNIT
FDB
FILE ADDRESS
NBLK
Fp
M
LU
LBLK
2
5
9 bits
Blockpointer
IPNT
AIPNT
(1) 0
SELF RELATIVE POINTER TO BLOCK
(2) 1
Block No
(3) 2
NBLK
(4) 3
LBLK
(5) 4
FILE ADDRESS
(6) 5
PROTECT BIT ADDRESS
(7) 6
NO. OF FIRST IN CORE BLOCK
(8) 7
NUMBER OF "POSSIBLE" IN CORE BLOCKS
(9) 8
R/W
Fp
M
I
F
UP
LU
(10) 9
ADDRESS FOR ERROR EXIT
File work area
IBUF
ABUF
BLOCKS OF THE FILE

DB UNIT DEFINITION Block

DB FILE DEFINITION Block

FILE PROTECT BIT

NUMBER IN CORE FDB'S

TE: THE ADDRESS OF FILE WORK AREA (IBUF) MAY BE CALCULATED AS:
ADDRESS (RELATIVE TO IPNT) = IPNT(1) - (IPNT(2) - IPNT(7)) *LBLK

it definition block the blocknumber of the
of the core res FDB for this unit plus the file number
the blocknumber of the first of core resident FDB's if
number less than NC. Similar remarks for the disk resident
s.

Fig. 1.

R/W READ/WRITE INDICATOR
(CONTROLS ADJUSTMENT OF IPNT(1) AND(7))

I IN CORE INDICATOR I.E. INFORMATION IN IBUF WAS READ IN

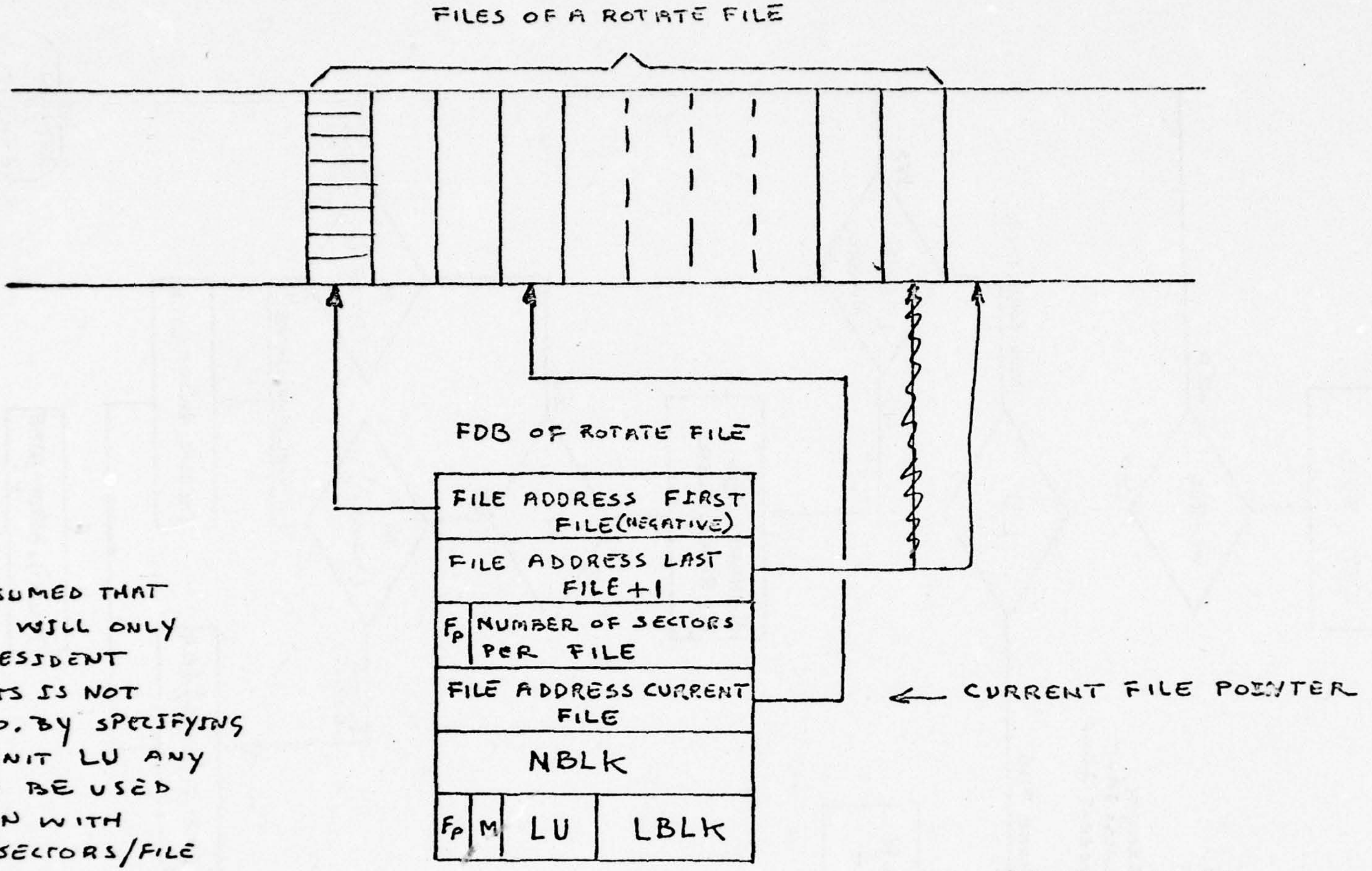
F FLAG TO INDICATE THAT A WRITE COMMAND STILL HAS TO BE PERFORMED

M WORD ADDRESSABLE MODE INDICATOR.

LU LOGICAL UNIT (5 BITS)

Fp FILE PROTECT BIT

UP bit to indicate "FROM UNPROTECTED PROGRAM"


FILES OF A ROTATE FILE
FDB OF ROTATE FILE
FILE ADDRESS FIRST FILE(NEGATIVE)
FILE ADDRESS LAST FILE+1
F_p
NUMBER OF SECTORS PER FILE
FILE ADDRESS CURRENT FILE
NBLK
F_p
M
LU
LBLK
← CURRENT FILE POINTER

NOTE IT IS ASSUMED THAT
ROTATE FILES WILL ONLY
BE DISK RESIDENT
HOWEVER THIS IS NOT
NECESSARY SO. BY SPECIFYING
THE LOGICAL UNIT LU ANY
DEVICE MAY BE USED
THE LOCATION WITH
NUMBER OF SECTORS/FILE
MAY THAN CONTAIN: NUMBER
OF WORDS OR NUMBER
OF blocks ETC/FILE.

F_p = FILE PROTECT BIT
M = WORD ADDRESSABLE MO
THE FILE ADDRESS OF THE FIRST
FILE IS COMPLEMENTED STORED
IN THIS WAY THE FDB IS TO BE
RECOGNIZED AS A SPECIAL ONE.

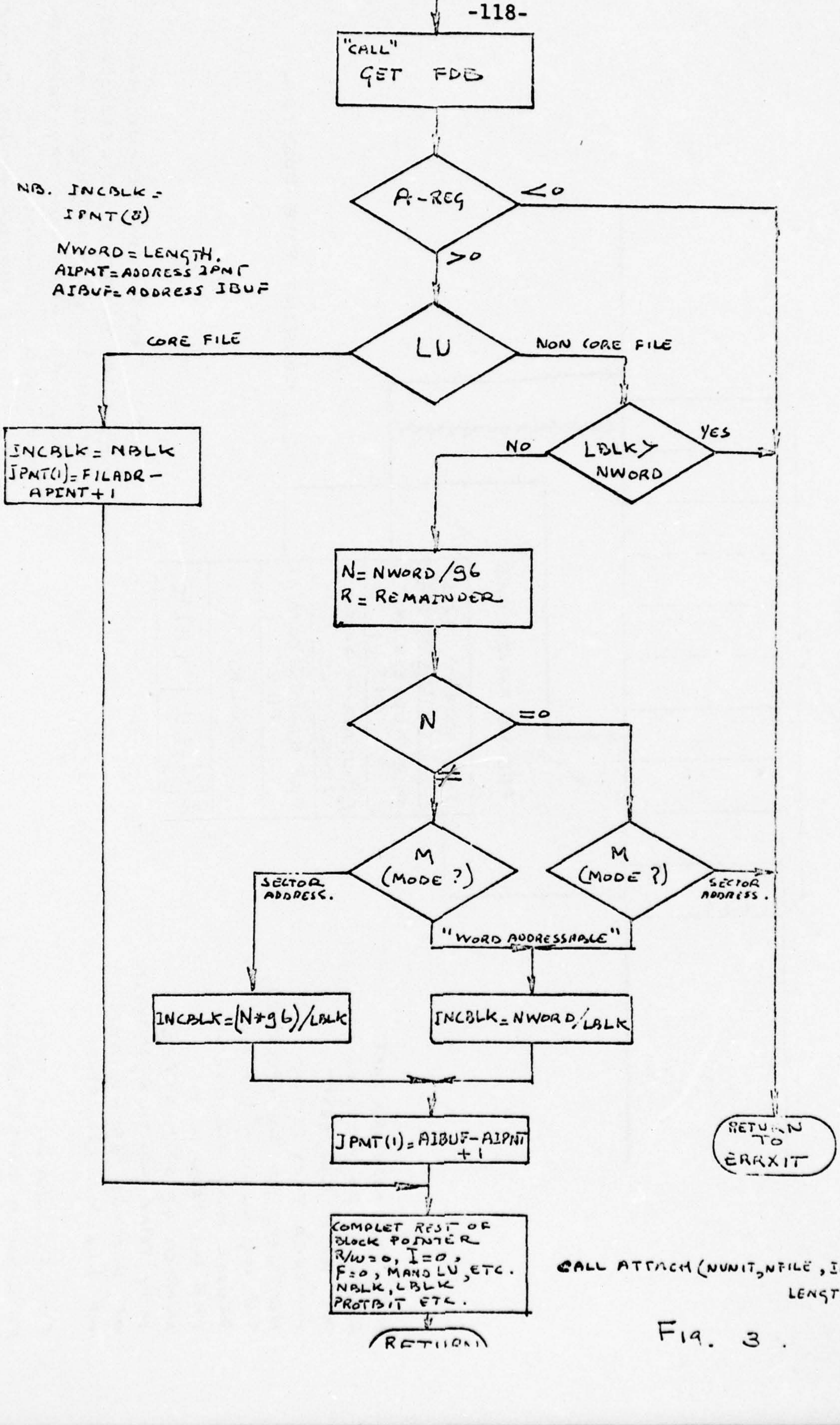
ROTATE FILE FDB'S

Fig. 2

NOTE THAT THE SECOND HALF
OF THE ROTATE FILE DEFINITION
BLOCK IS EXACTLY EQUAL TO A
REGULAR FDB AND MAY THEREFORE
BE USED AS SUCH.
IF THE FILE NUMBER OF THE ROTATE
FILE IS M, THE FILE NUMBER OF THE
SECOND HALF OF THE ROTATE FILE FDB IS
M+1.

NB. INCBLK = IPNT(8)

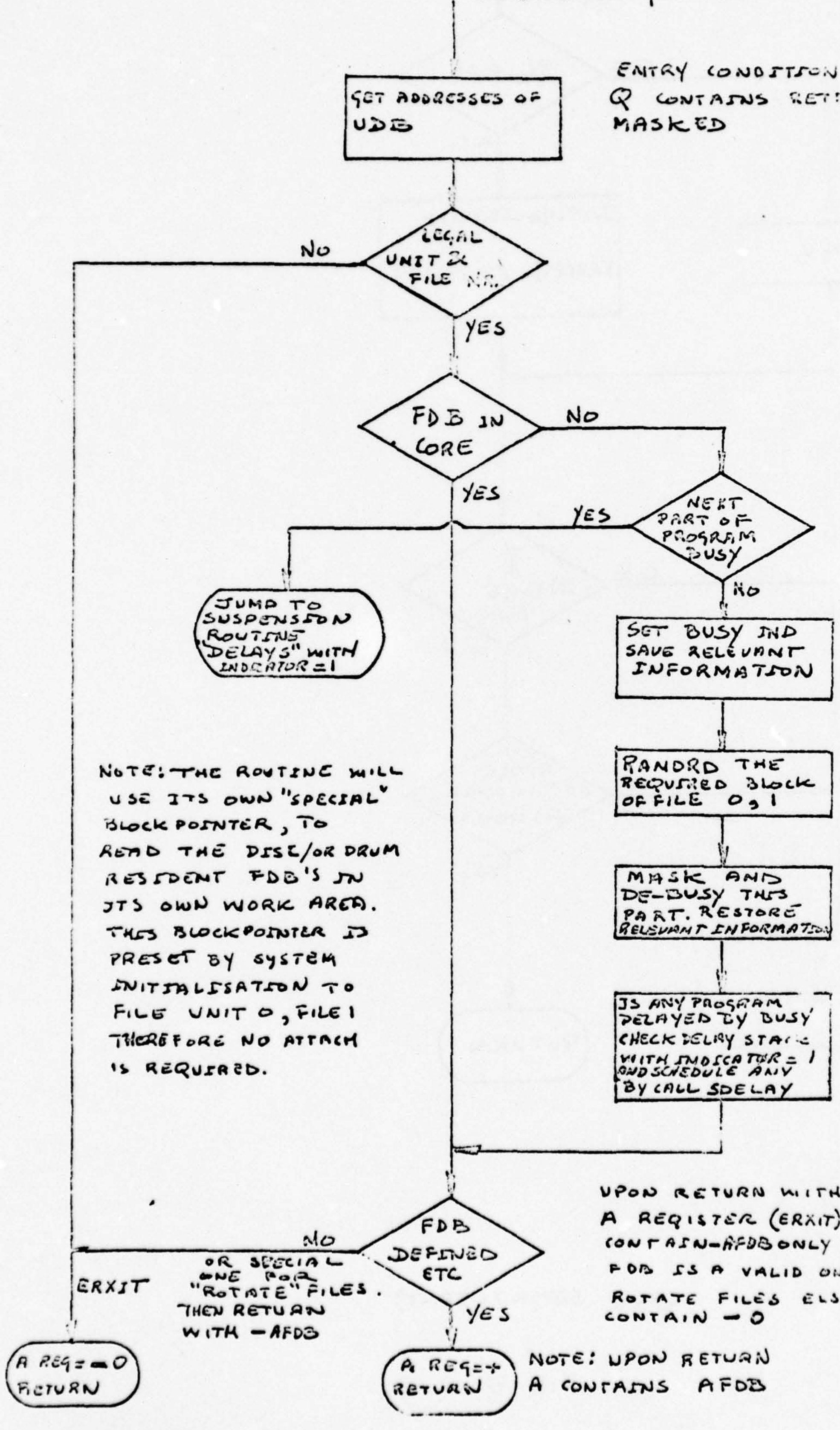
NWORD = LENGTH.
AIPNT = ADDRESS IPNT
AIBUF = ADDRESS IBUF


"CALL"
GET FDB
A-REG
<0
>0
LU
CORE FILE
NON CORE FILE
INCBLK = NBLK
IPNT(1) = FILADR - APINT + 1
LBLK > NWORD
NO
YES
N = NWORD/96
R = REMAINDER
N
=0
≠
M (MODE ?)
SECTOR ADDRESS.
M (MODE ?)
SECTOR ADDRESS.
"WORD ADDRESSABLE"
INCBLK = (N*96)/LBLK
INCBLK = NWORD/LBLK
IPNT(1) = AIBUF - AIPNT + 1
RETURN TO ERRXIT
COMPLET REST OF BLOCK POINTER
R/W = 0, I = 0,
F = 0, MAND LU, ETC.
NBLK, LBLK
PROTBIT ETC.
RETURN

CALL ATTACH (NUNIT, NFILE, IPNT, IBUF, LENGTH, ERRXIT)

Fig. 3.

"SUBROUTINE" GET FDB"


GET ADDRESSES OF UDB
NO
LEGAL UNIT & FILE NR.
YES
FDB IN CORE
NO
YES
YES
NEXT PART OF PROGRAM BUSY
NO
JUMP TO SUSPENSION ROUTINE "DELAYS" WITH INDICATOR = 1
SET BUSY IND SAVE RELEVANT INFORMATION
RANDRD THE REQUIRED BLOCK OF FILE 0, 1
MASK AND DE-BUSY THIS PART. RESTORE RELEVANT INFORMATION
IS ANY PROGRAM DELAYED BY BUSY CHECK DELAY STACK WITH INDICATOR = 1 AND SCHEDULE ANY BY CALL SDELAY
FDB DEFINED ETC
NO
ERXIT
YES
A REQ = −0 RETURN
A REQ = + RETURN

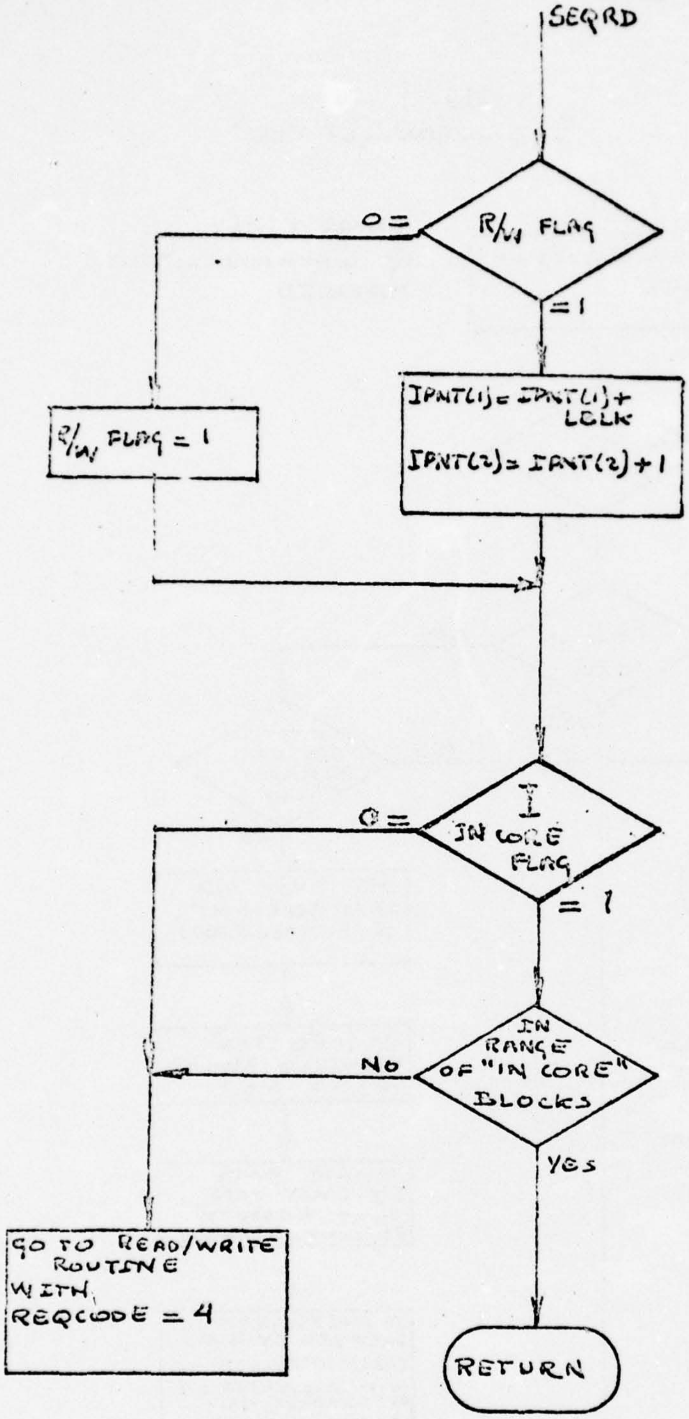
ENTRY CONDITION
Q CONTAINS RETUSE
MASKED

NOTE: THE ROUTINE WILL USE ITS OWN "SPECIAL" BLOCK POINTER, TO READ THE DISC/OR DRUM RESIDENT FDB'S IN ITS OWN WORK AREA. THIS BLOCKPOINTER IS PRESET BY SYSTEM INITIALISATION TO FILE UNIT 0, FILE 1 THEREFORE NO ATTACH IS REQUIRED.

OR SPECIAL ONE FOR "ROTATE" FILES. THEN RETURN WITH −AFDB

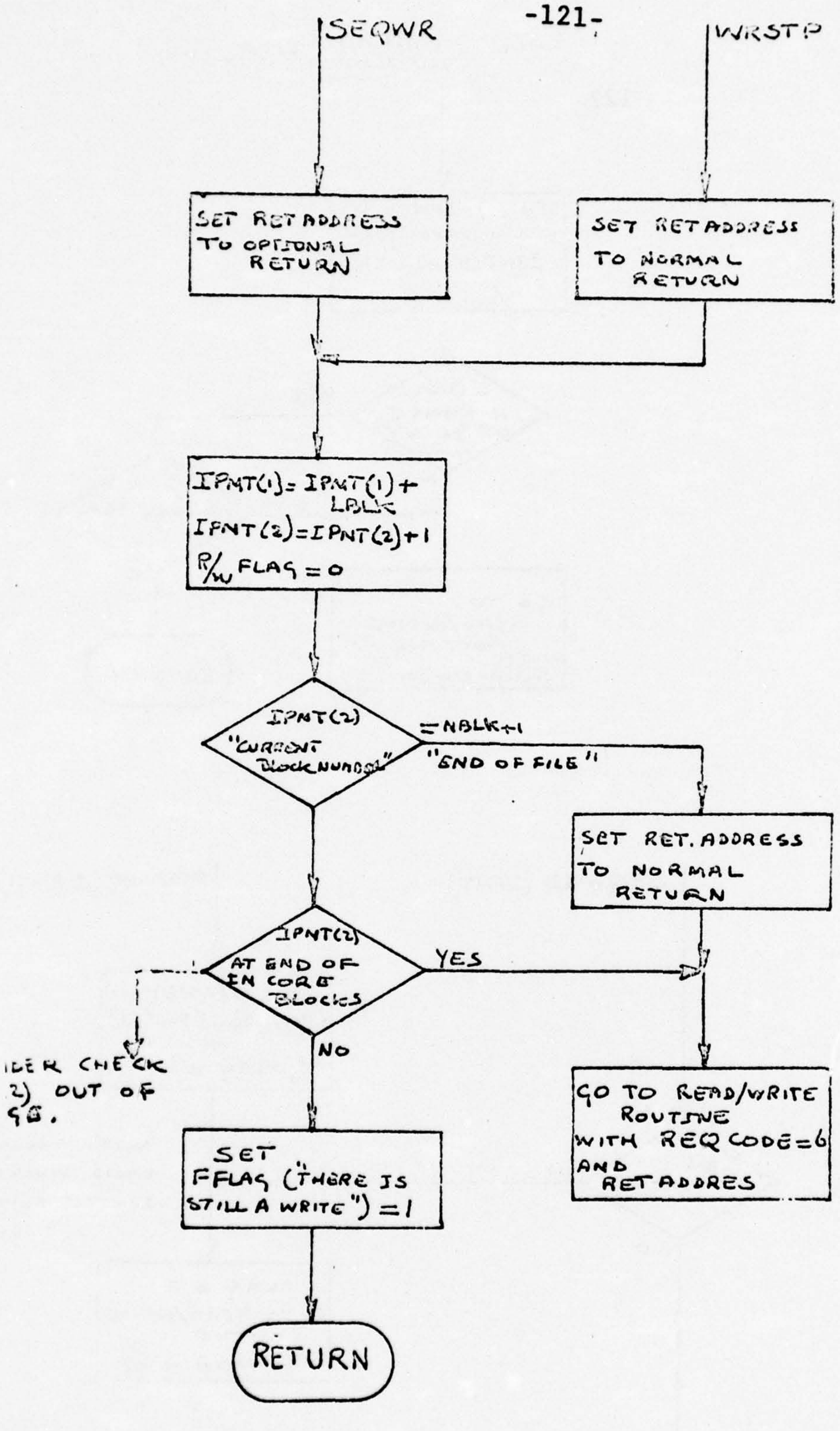
UPON RETURN WITH NEGATIVE A REGISTER (ERXIT) A SCHOULD CONTAIN −AFDB ONLY IF THE FDB IS A VALID ONE FOR ROTATE FILES ELSE IT SCHOULD CONTAIN −0

NOTE: UPON RETURN A CONTAINS AFDB


SEQRD
R/W FLAG
0 =
= 1
R/W FLAG = 1
IPNT(1) = IPNT(1) + LBLK
IPNT(2) = IPNT(2) + 1
I IN CORE FLAG
0 =
= 1
IN RANGE OF "IN CORE" BLOCKS
NO
YES
GO TO READ/WRITE ROUTINE WITH REQCODE = 4
RETURN

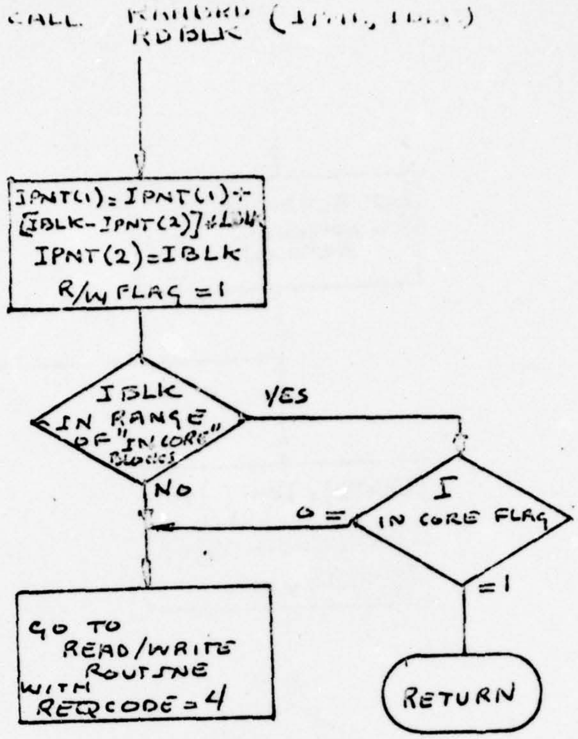
CALL SEQRD (IPNT)

Fig 5


SEQWR
WRSTP
SET RET ADDRESS TO OPTIONAL RETURN
SET RET ADDRESS TO NORMAL RETURN
IPNT(1)= IPNT(1)+ LBLK
IPNT(2)=IPNT(2)+1
R/W FLAG = 0
IPNT(2) "CURRENT BLOCK NUMBER"
=NBLK+1
"END OF FILE"
SET RET. ADDRESS TO NORMAL RETURN
IPNT(2) AT END OF IN CORE BLOCKS
YES
NO
IDER CHECK
2) OUT OF
GE.
GO TO READ/WRITE ROUTINE WITH REQ CODE=6 AND RET ADDRES
SET FFLAG ("THERE IS STILL A WRITE") = 1
RETURN

CALL SEQWR (IPNT, RETURN)
CALL WRSTP (IPNT)

Fig. 6


CALL
RDBLK
IPNT(1)=IPNT(1)+
[IBLK-IPNT(2)]*LBLK
IPNT(2)=IBLK
R/W FLAG =1
IBLK
IN RANGE
OF "IN CORE"
BLOCKS
YES
NO
I
IN CORE FLAG
0 =
=1
GO TO
READ/WRITE
ROUTINE
WITH
REQCODE =4
RETURN

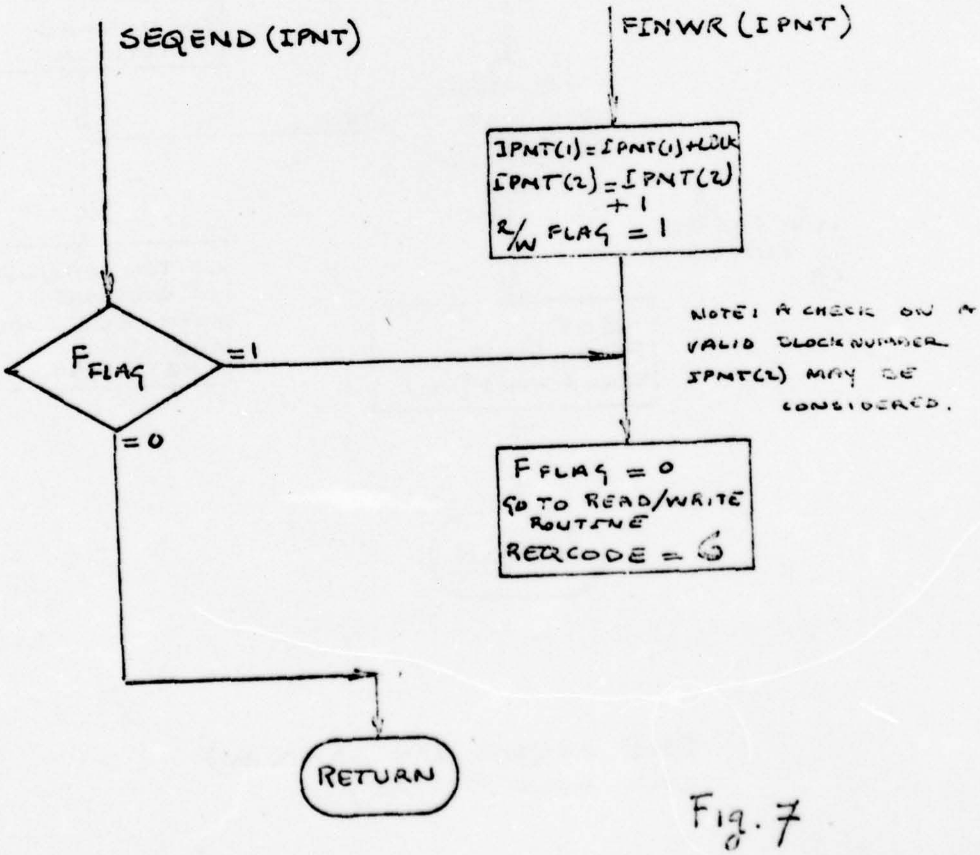
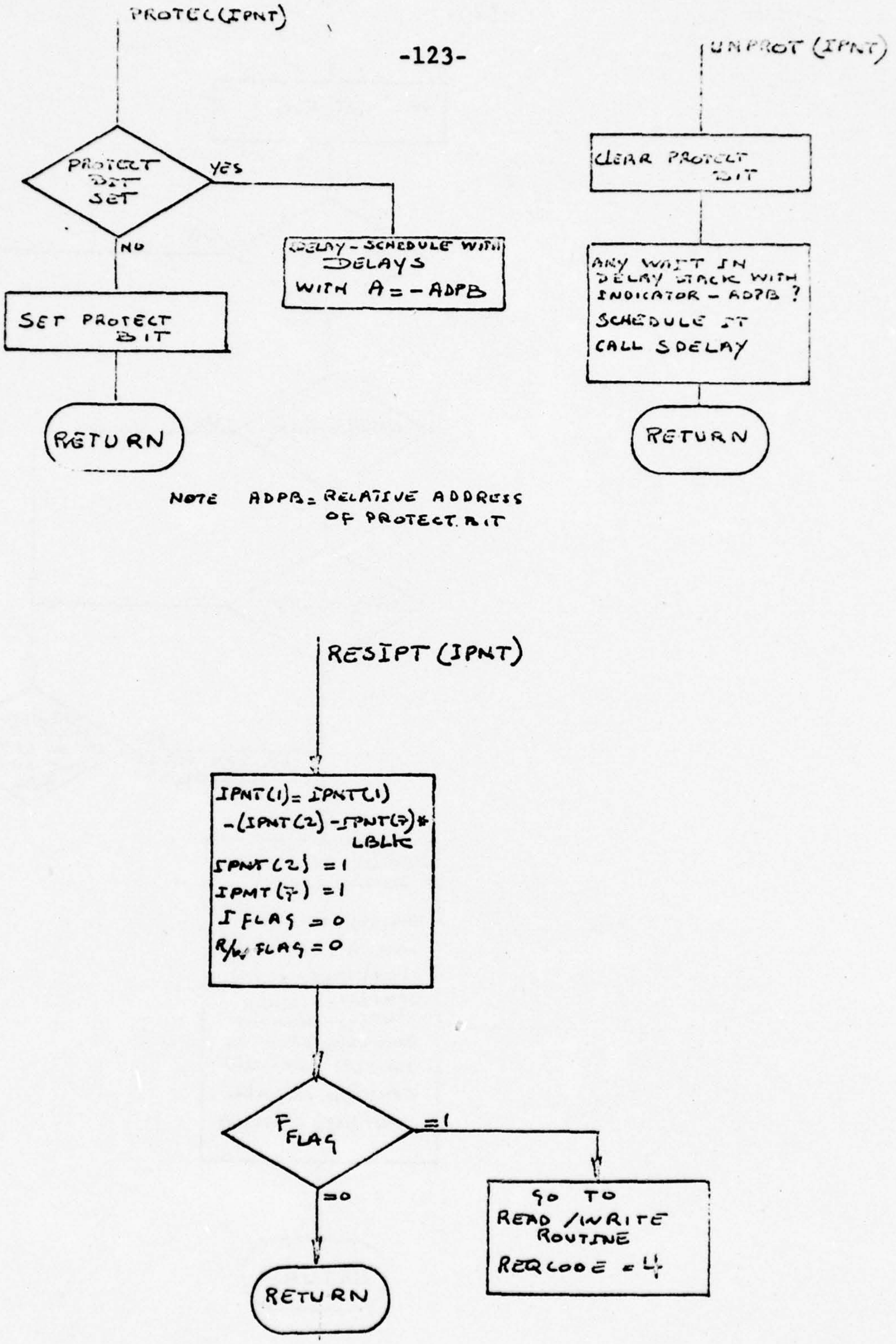
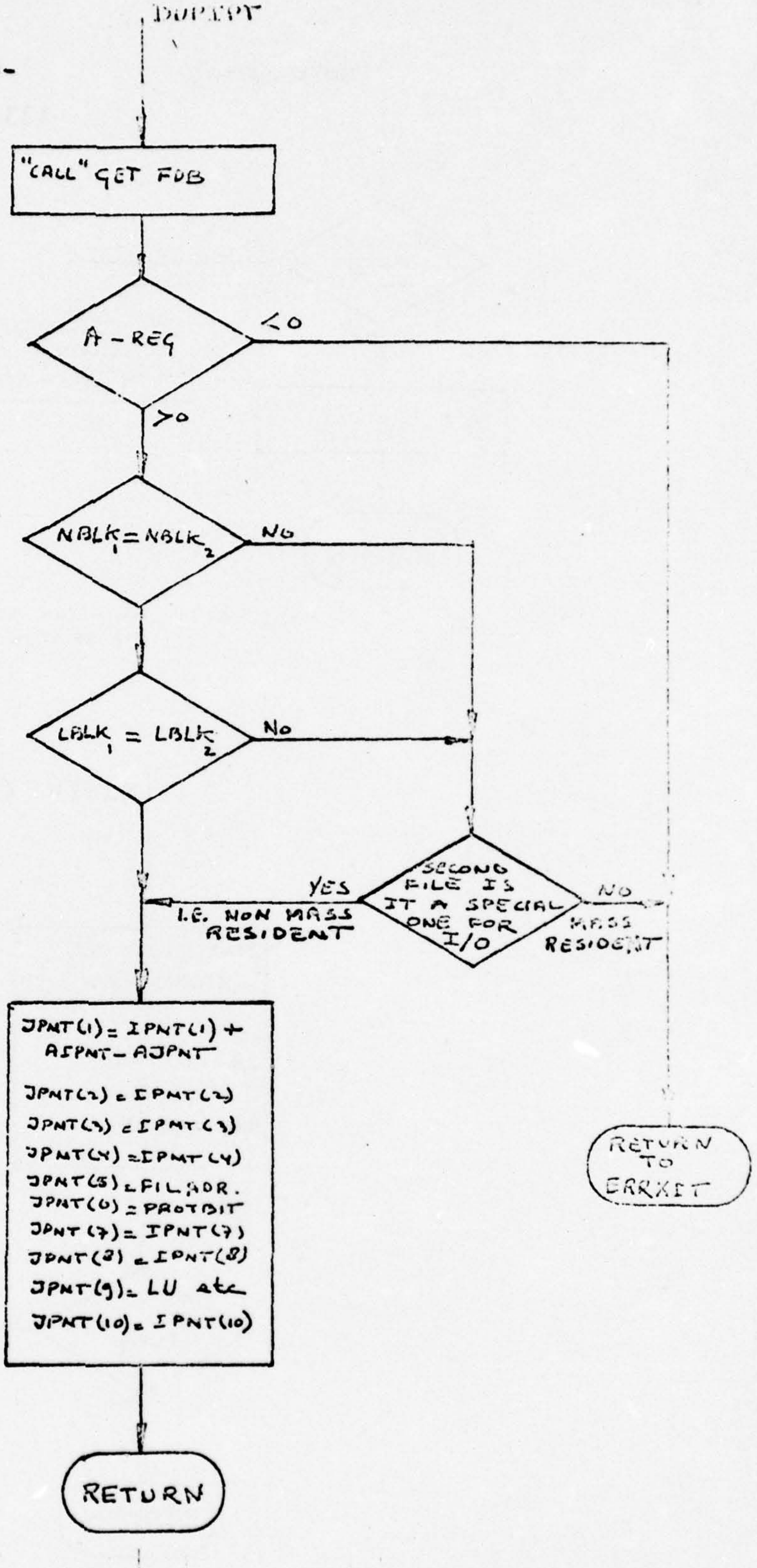

SEQEND (IPNT)
FINWR (IPNT)
IPNT(1)=IPNT(1)+LBLK
IPNT(2)=IPNT(2)
+1
R/W FLAG =1
FFLAG
=1
=0
NOTE: A CHECK ON A
VALID BLOCK NUMBER
IPNT(2) MAY BE
CONSIDERED.
FFLAG = 0
GO TO READ/WRITE
ROUTINE
REQCODE = 6
RETURN

Fig. 7


PROTEC(IPNT)
PROTECT BIT SET
YES
NO
SET PROTECT BIT
SDELAY - SCHEDULE WITH DELAYS WITH A = - ADPB
RETURN
UNPROT (IPNT)
CLEAR PROTECT BIT
ANY WAIT IN DELAY STACK WITH INDICATOR - ADPB ?
SCHEDULE IT
CALL SDELAY
RETURN
RESIPT (IPNT)
IPNT(1) = IPNT(1) - (IPNT(2) - IPNT(7))* LBLK
IPNT(2) = 1
IPMT(7) = 1
IFLAG = 0
R/W FLAG = 0
FFLAG
=1
=0
GO TO READ /WRITE ROUTINE
REQCODE = 4
RETURN

NOTE ADPB = RELATIVE ADDRESS OF PROTECT BIT

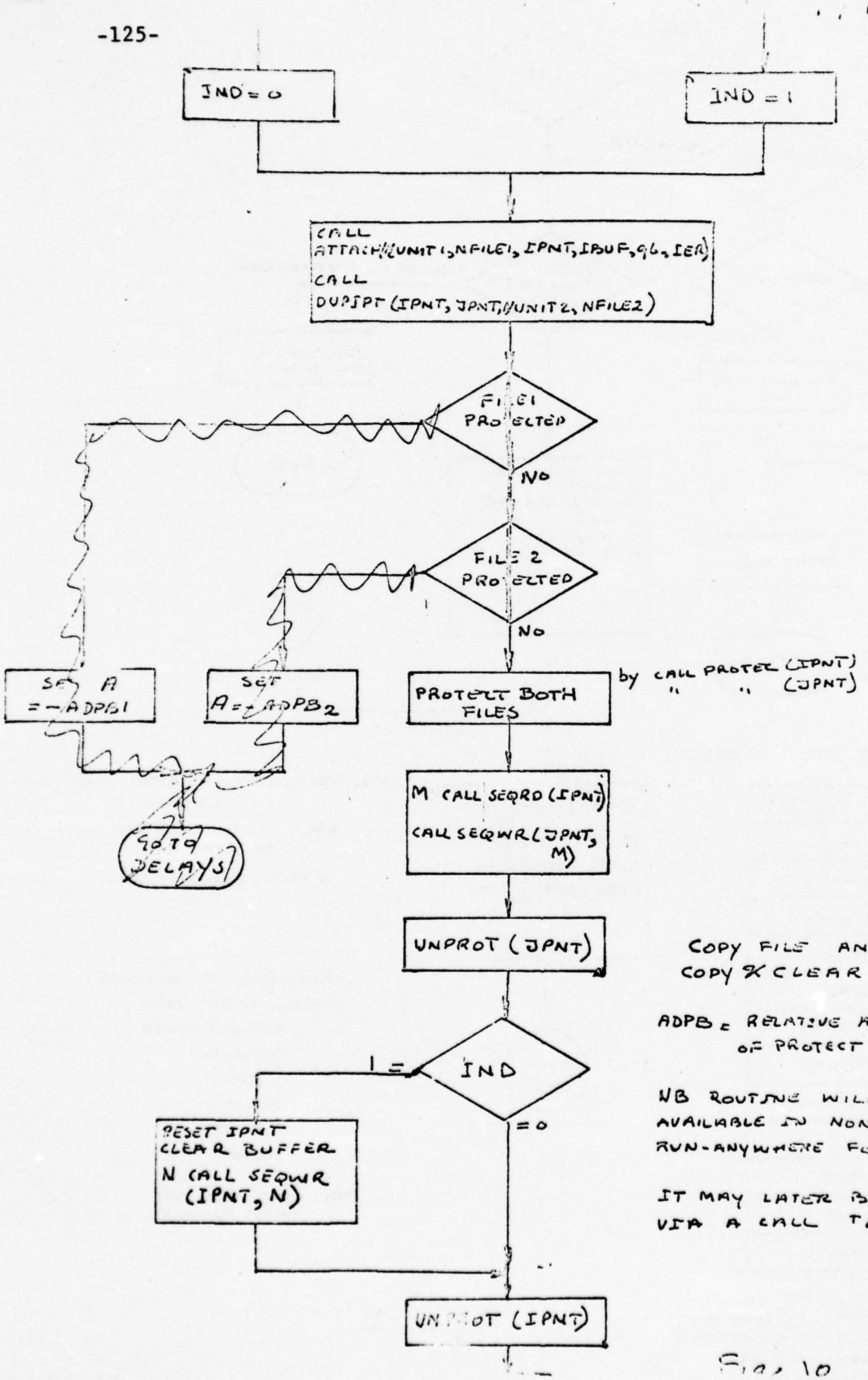
Fig. 8.


DUPIPT
"CALL" GET FDB
A-REG
<0
>0
NBLK1 = NBLK2
NO
LBLK1 = LBLK2
No
SECOND FILE IS IT A SPECIAL ONE FOR I/O
YES
I.E. NON MASS RESIDENT
NO
MASS RESIDENT
JPNT(1) = IPNT(1) + ASPNT - AJPNT
JPNT(2) = IPMT(2)
JPNT(3) = IPMT(3)
JPMT(4) = IPMT(4)
JPNT(5) = FIL.ADR.
JPNT(6) = PROTBIT
JPNT(7) = IPNT(7)
JPNT(8) = IPNT(8)
JPNT(9) = LU etc
JPNT(10) = IPNT(10)
RETURN
RETURN TO ERRXIT

CALL DUPIPT (IPNT, JPNT, NUNIT, NFILE)

Fig. 9

4


IND = 0
IND = 1
CALL
ATTACH(//UNIT1, NFILE1, IPNT, IBUF, 96, IER)
CALL
DUPSPT (IPNT, JPNT, //UNIT2, NFILE2)
FILE1 PROTECTED
NO
FILE 2 PROTECTED
NO
SET A = −ADPB1
SET A = −ADPB2
PROTECT BOTH FILES
by CALL PROTEC (IPNT)
" " (JPNT)
GO TO DELAYS
M CALL SEQRD (IPNT)
CALL SEQWR (JPNT, M)
UNPROT (JPNT)
IND
1 =
= 0
RESET IPNT
CLEAR BUFFER
N CALL SEQWR (IPNT, N)
UNPROT (IPNT)

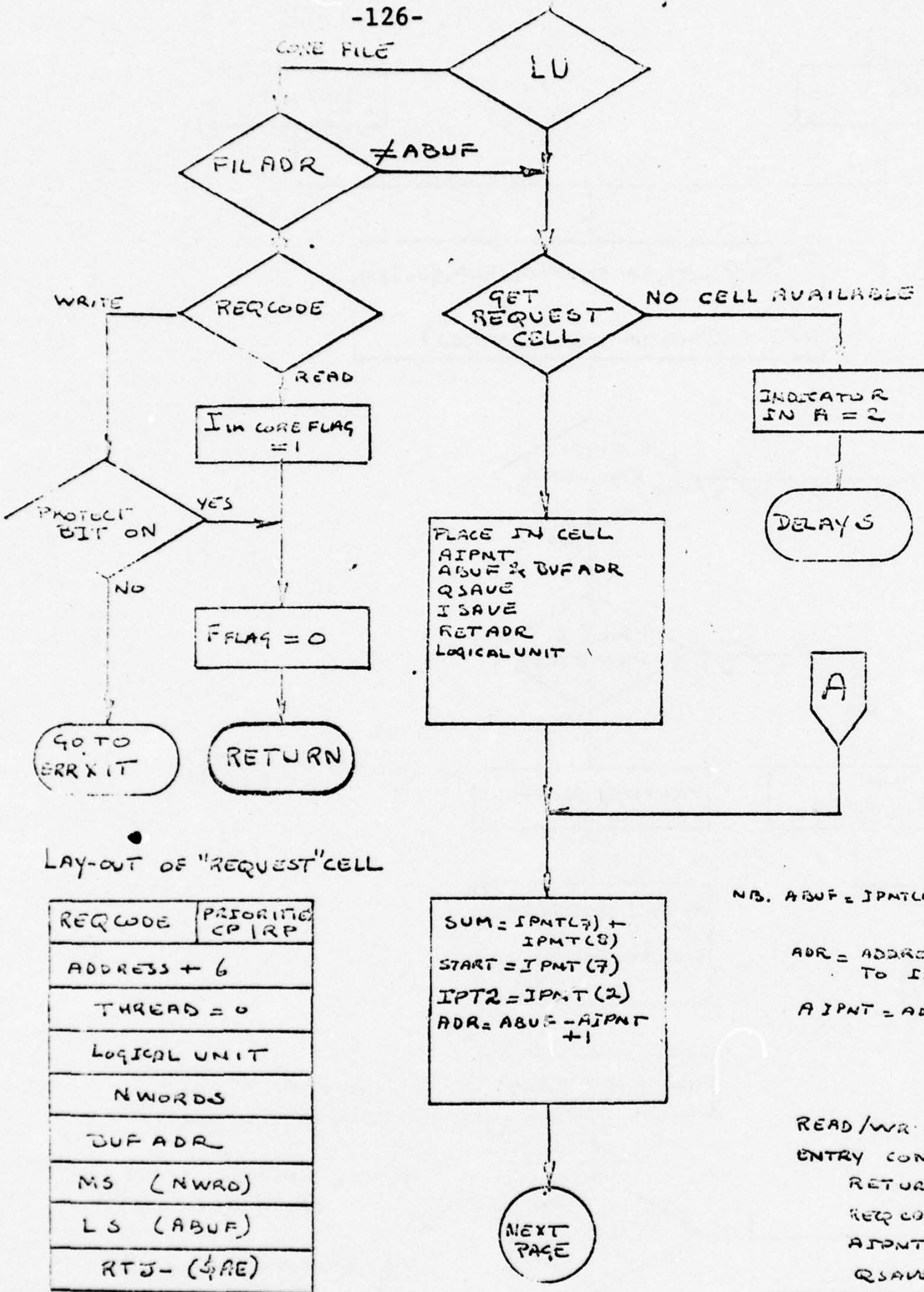
COPY FILE AND
COPY & CLEAR FILE

ADPB = RELATIVE ADDRESS
OF PROTECT BIT

NB ROUTINE WILL BE AVAILABLE IN NON-REENTRANT RUN-ANYWHERE FORM.

IT MAY LATER BE CALLED VIA A CALL TO LOCAL.

Fig. 10


LU
CORE FILE
FILADR
≠ABUF
REQCODE
WRITE
READ
I in CORE FLAG = 1
PROTECT BIT ON
YES
NO
FFLAG = 0
GO TO ERRXIT
RETURN
GET REQUEST CELL
NO CELL AVAILABLE
INDICATOR IN A = 2
DELAYS
PLACE IN CELL
AIPNT
ABUF & BUFADR
QSAVE
ISAVE
RETADR
LOGICALUNIT
A
SUM = IPNT(7) + IPNT(8)
START = IPNT(7)
IPT2 = IPNT(2)
ADR = ABUF - AIPNT + 1
NEXT PAGE

NB. ABUF = IPNT(1) - (IPNT(2) - IPNT(7)) [illegible]
+ AIPNT - 1

ADR = ADDRESS BUFFER RELATIVE TO IPNT

AIPNT = ADDRESS IPNT

LAY-OUT OF "REQUEST" CELL

REQCODE	PRIORITIES CP IRP
ADDRESS + 6	
THREAD = 0	
LOGICAL UNIT	
NWORDS	
BUF ADR	
MS (NWRD)	
LS (ABUF)	
RTJ- (¼AE)	
AIPNT	
QSAVE	
ISAVE	
RET ADR	

NOTE: THE REQUEST CELL CONTAINS IN PRINCIPLE THE ARGUMENTS (FIRST 8 LOCATIONS) FOR A MONITOR REQUEST FOR INPUT/OUTPUT

READ/WRITE ROUTINE
ENTRY CONDITIONS:
- RETURN ADDRE..
- REQCODE
- AIPNT
- QSAVE
- ISAVE
- ABUF

NB. CORE FILES ARE ALWAYS "WORD ADDRESSABLE"

Fig 11 a

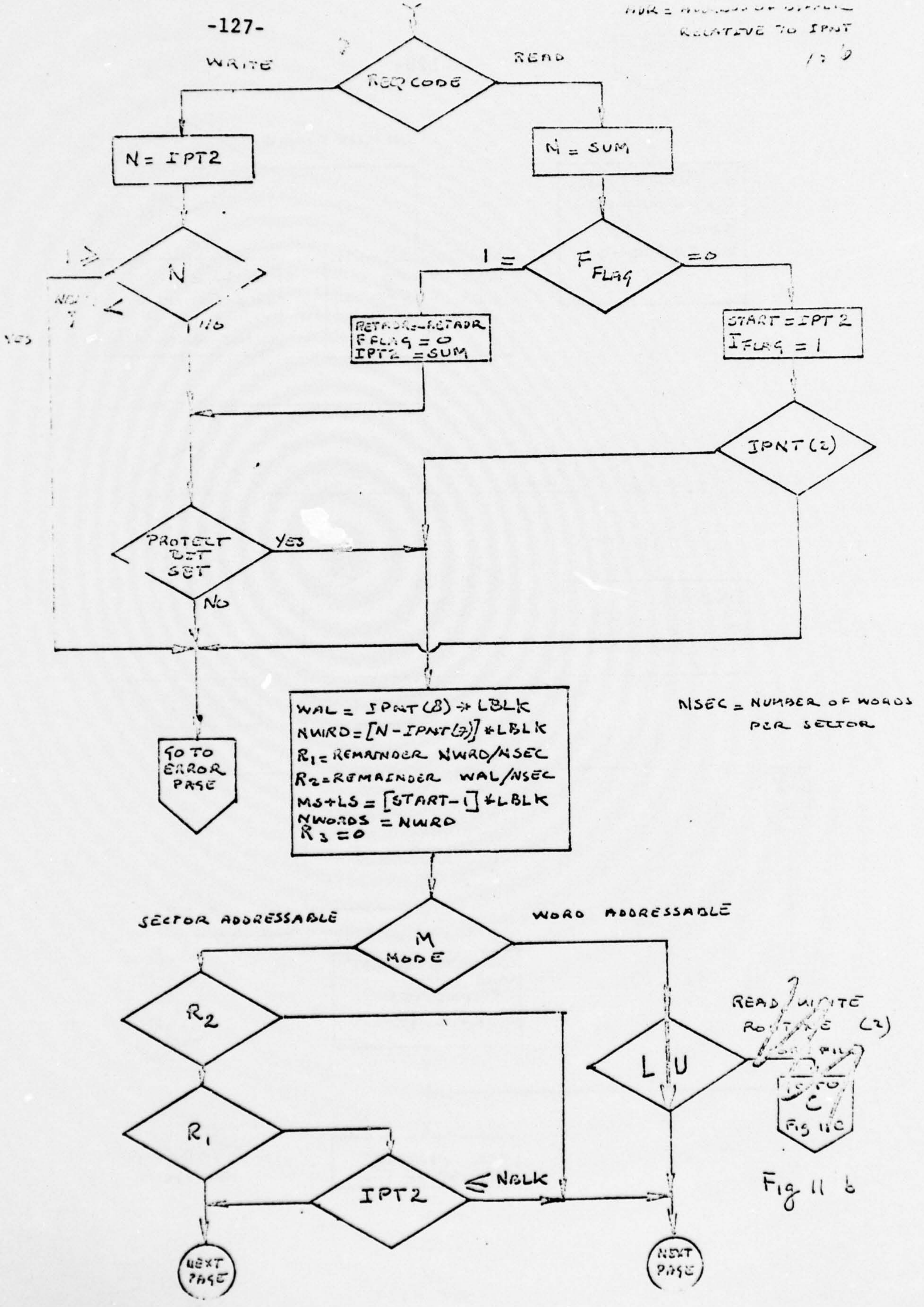
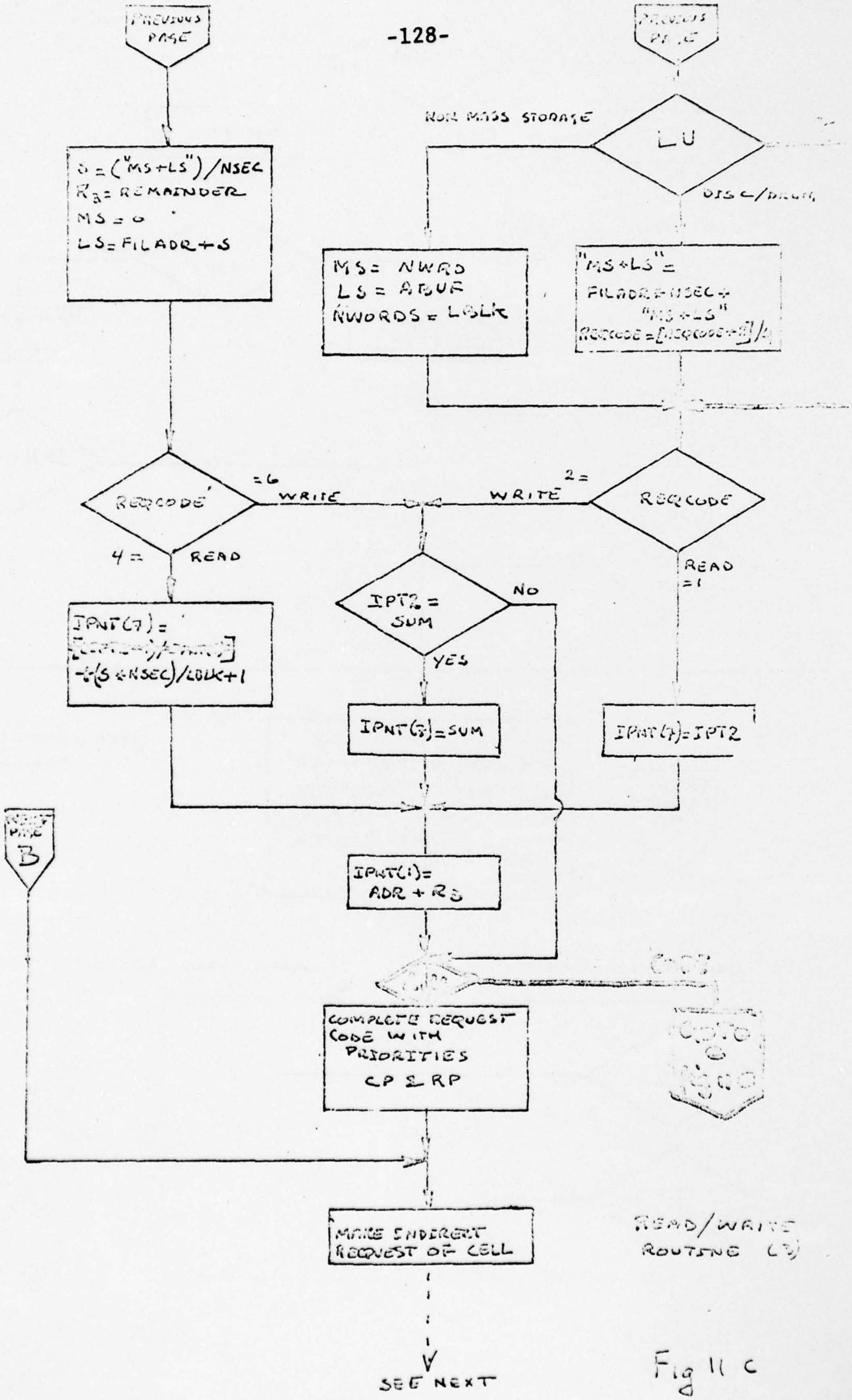
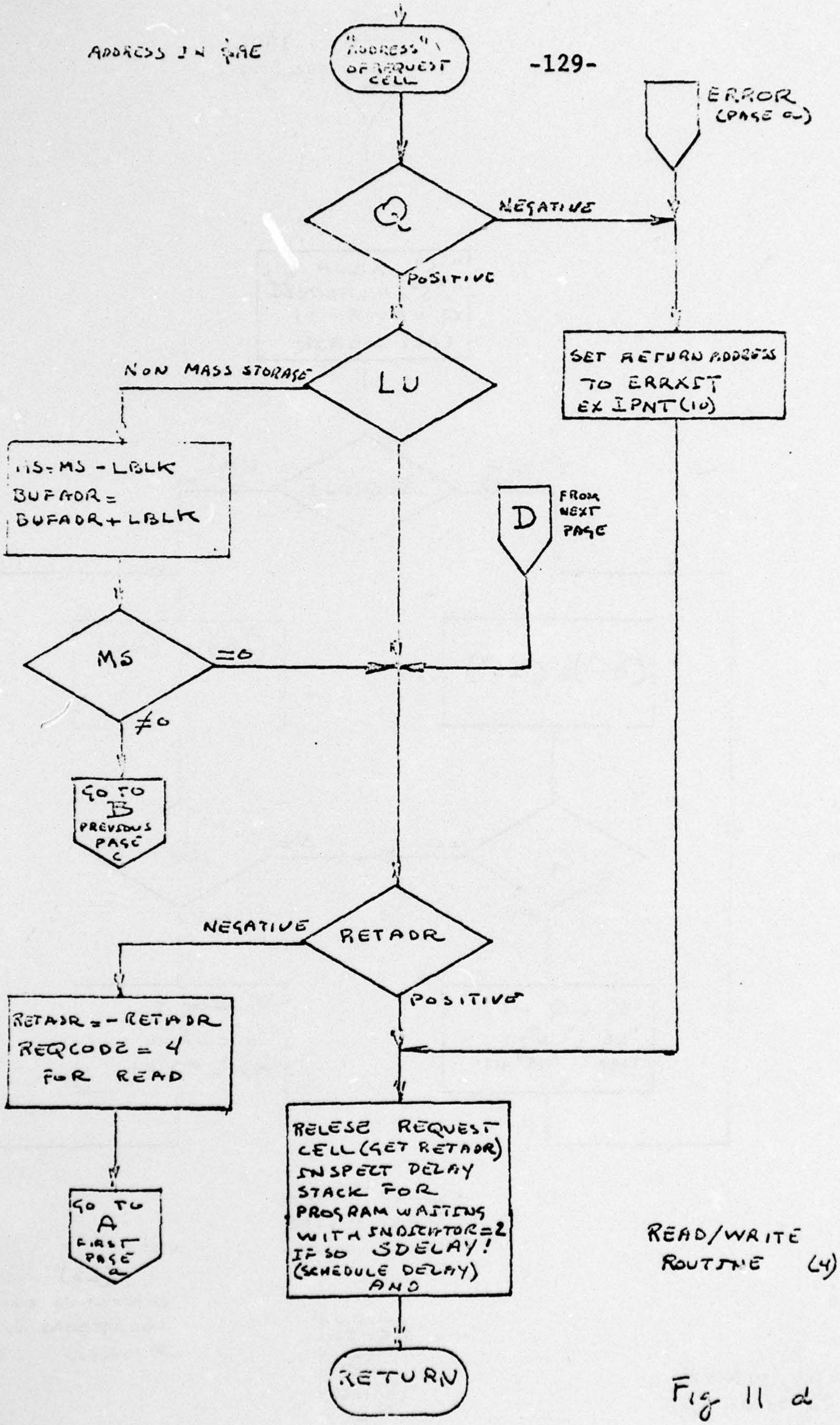

ADR = [illegible] RELATIVE TO IPNT
1 : b
WRITE
READ
REQ CODE
N = IPT2
N = SUM
N
1 >
NO
<
YES
NO
1 =
F FLAG
= 0
RETADR = RETADR
FFLAG = 0
IPT2 = SUM
START = IPT 2
IFLAG = 1
IPNT(2)
PROTECT BIT SET
YES
NO
GO TO ERROR PAGE
WAL = IPNT(8) * LBLK
NWRD = [N - IPNT(7)] * LBLK
R1 = REMAINDER NWRD/NSEC
R2 = REMAINDER WAL/NSEC
MS+LS = [START - 1] * LBLK
NWORDS = NWRD
R3 = 0
NSEC = NUMBER OF WORDS PER SECTOR
SECTOR ADDRESSABLE
WORD ADDRESSABLE
M MODE
R2
R1
IPT2
≤ NBLK
L U
READ WRITE ROUTINE (2)
GO TO C Fig 11C
NEXT PAGE
NEXT PAGE

Fig 11 b


PREVIOUS PAGE
PREVIOUS PAGE
NON MASS STORAGE
LU
DISC/DRUM
S = ("MS+LS")/NSEC
R_3 = REMAINDER
MS = 0
LS = FILADR+S
MS = NWRD
LS = ABUF
NWORDS = LBLK
"MS+LS" =
FILADR+NSEC+
"MS+LS"
REQCODE = [REQCODE+[illegible]]/4
REQCODE
=6
WRITE
4 =
READ
WRITE
2 =
REQCODE
READ
=1
IPT2 =
SUM
NO
YES
IPNT(7) =
[illegible]
-(S+NSEC)/LBLK+1
IPNT(3)=SUM
IPNT(7)=IPT2
IPNT(1)=
ADR + R_3
COMPLETE REQUEST
CODE WITH
PRIORITIES
CP & RP
B
MAKE INDIRECT
REQUEST OF CELL
SEE NEXT

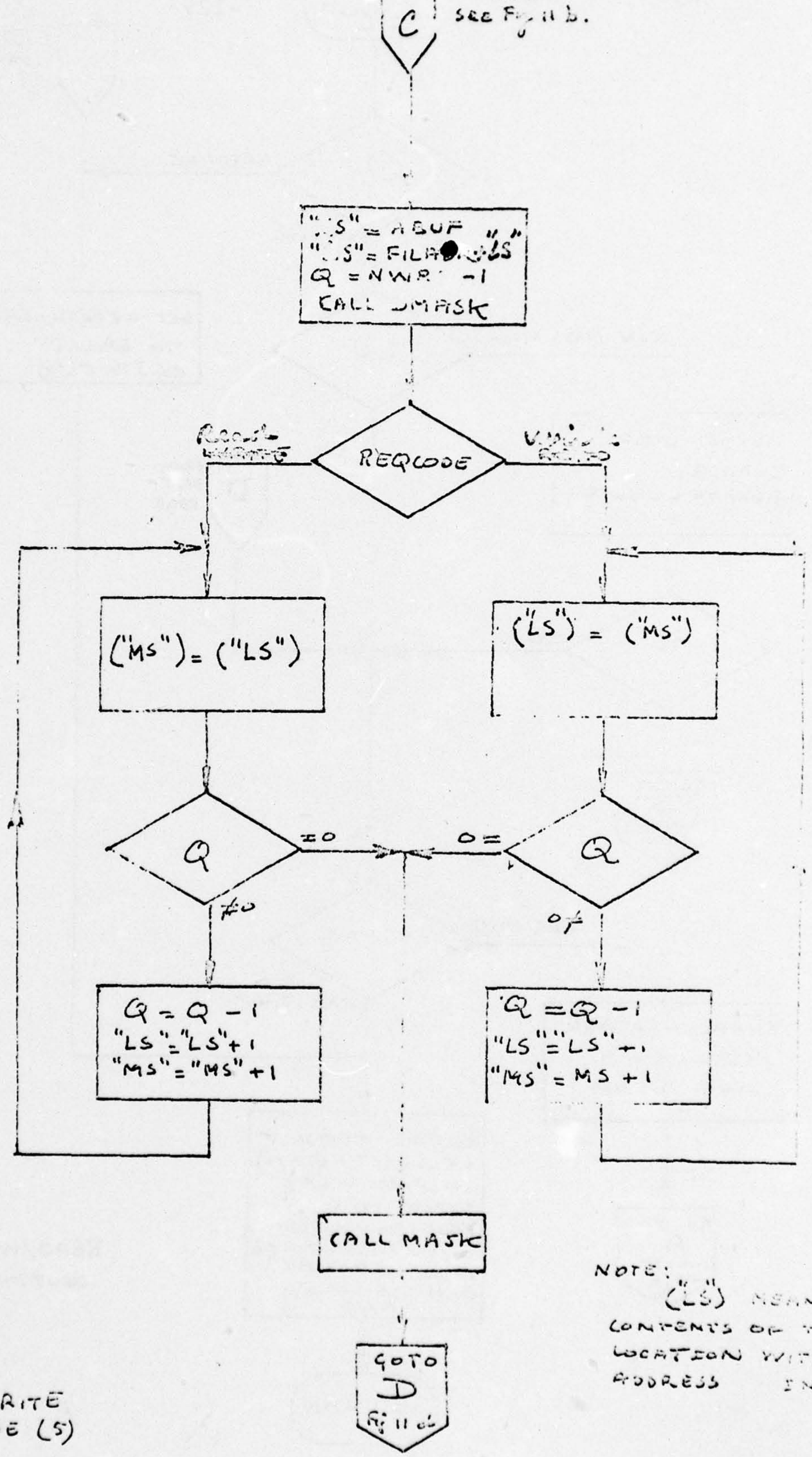
READ/WRITE
ROUTINE (3)

Fig 11 c


ADDRESS IN PAGE
"ADDRESS" OF REQUEST CELL
Q
NEGATIVE
POSITIVE
ERROR (PAGE a)
LU
NON MASS STORAGE
SET RETURN ADDRESS TO ERRXST EX IPNT(10)
MS=MS-LBLK
BUFADR=
BUFADR+LBLK
D
FROM NEXT PAGE
MS
=0
≠0
GO TO B PREVIOUS PAGE c
RETADR
NEGATIVE
POSITIVE
RETADR=-RETADR
REQCODE = 4
FOR READ
GO TO A FIRST PAGE a
RELESE REQUEST CELL (GET RETADR) INSPECT DELAY STACK FOR PROGRAM WAITING WITH INDICATOR=2 IF SO SDELAY! (SCHEDULE DELAY) AND
RETURN

READ/WRITE ROUTINE (4)

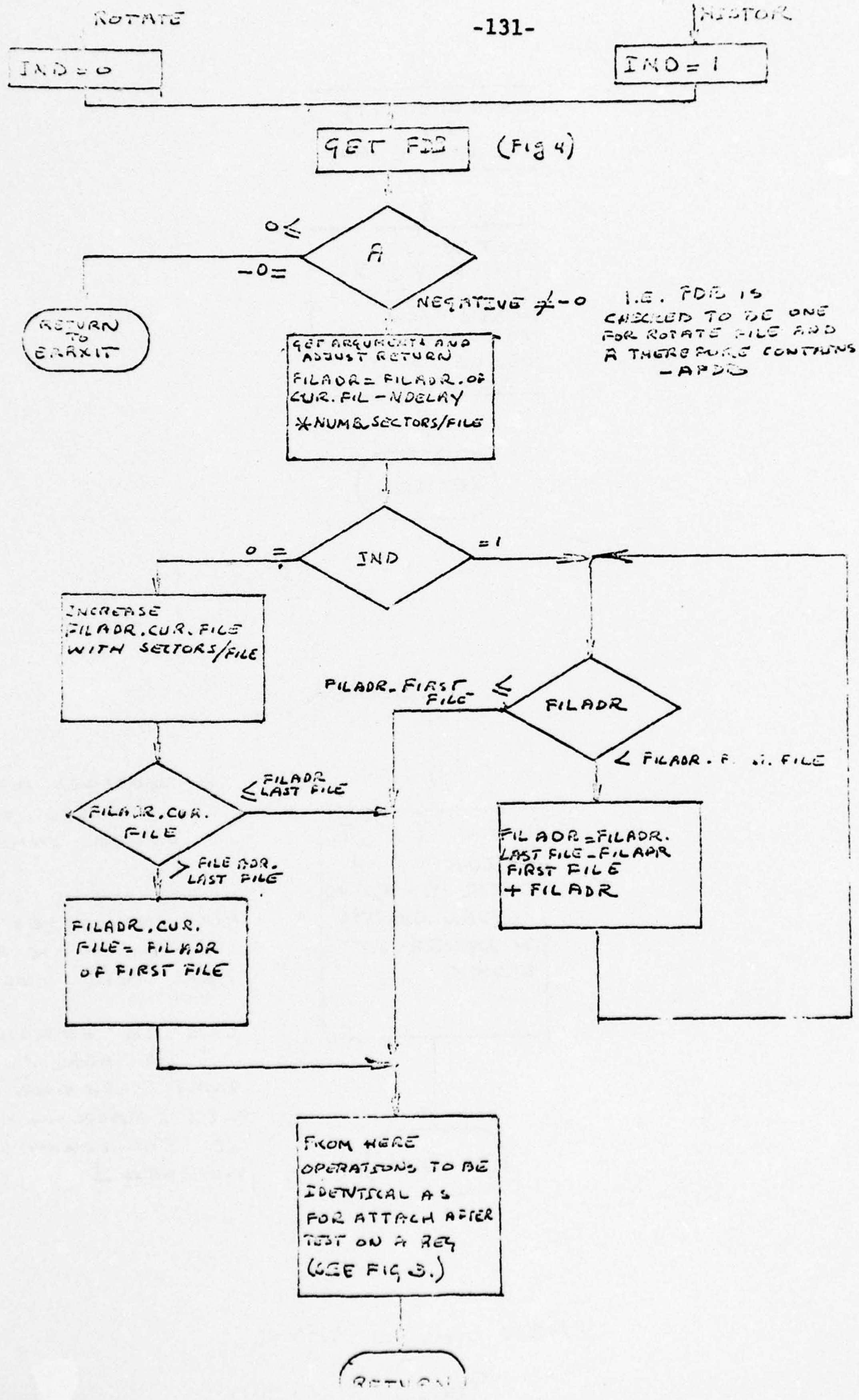
Fig 11 d

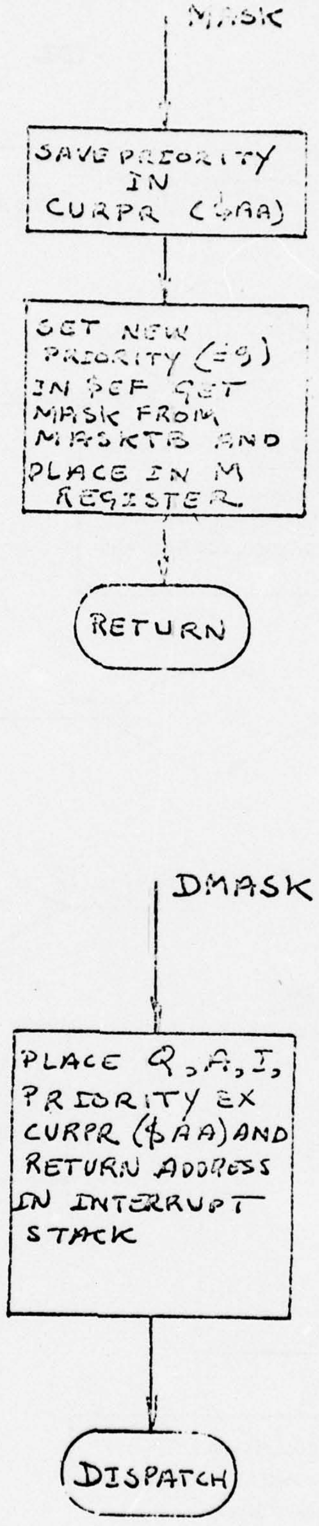

C
see Fig 11 b.
"LS" = ABUF
"MS" = FILA[illegible] "LS"
Q = NWR[illegible] -1
CALL DMASK
REQCODE
Read
Write
("MS") = ("LS")
("LS") = ("MS")
Q
Q
=0
0=
≠0
0≠
Q = Q - 1
"LS" = "LS" + 1
"MS" = "MS" + 1
Q = Q - 1
"LS" = "LS" + 1
"MS" = MS + 1
CALL MASK
GOTO
D
Fig 11 d

NOTE:
("LS") MEANS
CONTENTS OF THE
LOCATION WITH ITS
ADDRESS IN "LS"

READ/WRITE
ROUTINE (5)

Fig. 11 e


ROTATE
HISTOR
IND = 0
IND = 1
GET FDB
(Fig 4)
A
0≤
-0=
NEGATIVE ≠-0
I.E. FDB IS CHECKED TO BE ONE FOR ROTATE FILE AND A THEREFORE CONTAINS -AFDB
RETURN TO ERRXIT
GET ARGUMENTS AND ADJUST RETURN
FILADR= FILADR. OF CUR. FIL - NDELAY *NUMB. SECTORS/FILE
IND
0 =
=1
INCREASE FILADR.CUR.FILE WITH SECTORS/FILE
FILADR. FIRST FILE ≤
FILADR
< FILADR. F. .I. FILE
FILADR.CUR. FILE
≤ FILADR LAST FILE
> FILE ADR. LAST FILE
FILADR.CUR. FILE= FILADR OF FIRST FILE
FIL ADR =FILADR. LAST FILE - FILADR FIRST FILE + FILADR
FROM HERE OPERATIONS TO BE IDENTICAL AS FOR ATTACH AFTER TEST ON A REG (SEE FIG 3.)
RETURN


MASK
SAVE PRIORITY IN CURPR ($AA)
GET NEW PRIORITY (E9) IN $EF GET MASK FROM MASKTB AND PLACE IN M REGISTER
RETURN
DMASK
PLACE Q, A, I, PRIORITY EX CURPR ($AA) AND RETURN ADDRESS IN INTERRUPT STACK
DISPATCH

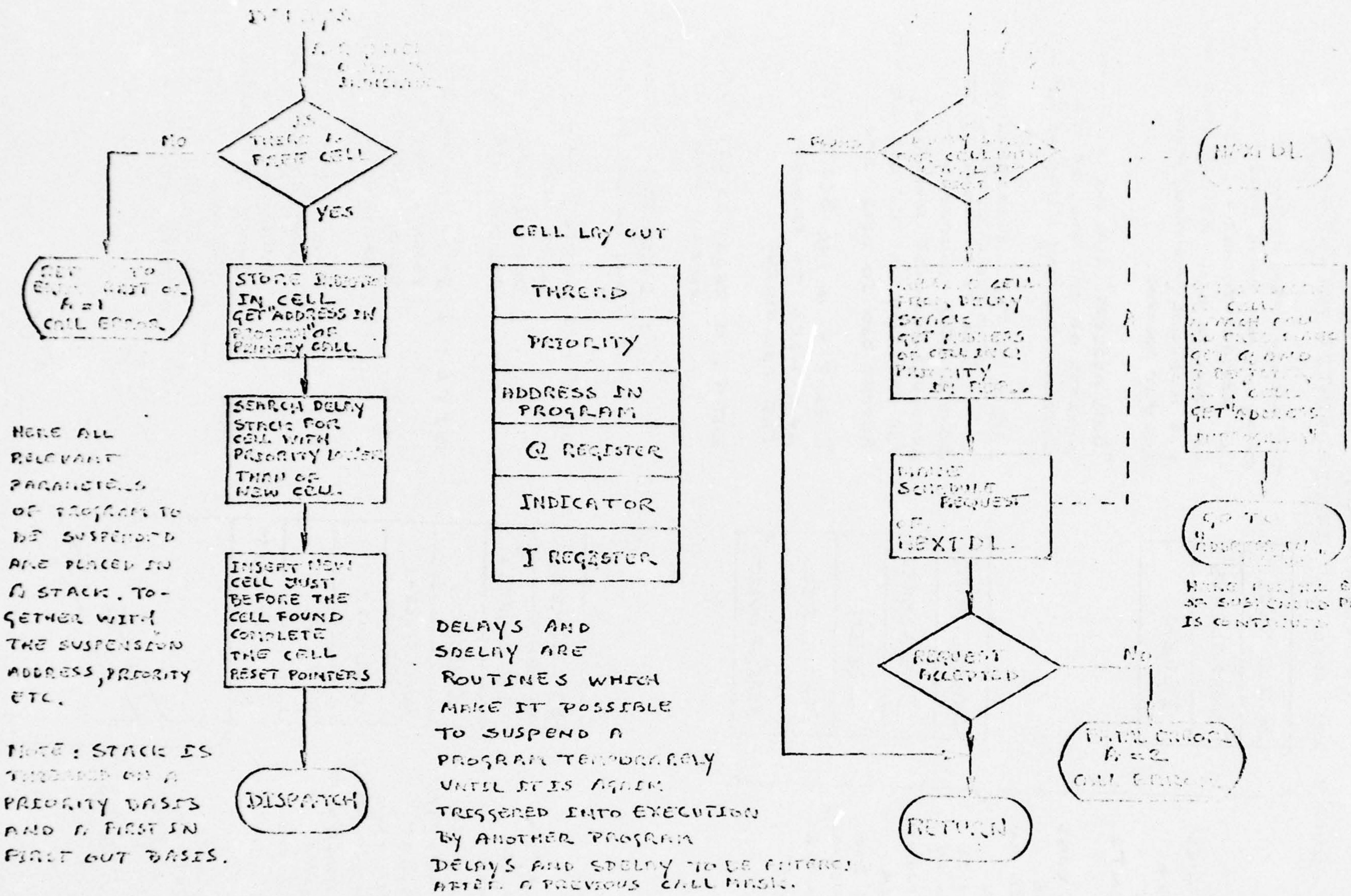
NB. ROUTINES ARE
OF COURSE EXECUTED
UNDER INHIBIT INTER.

WHEN MASK IS USED
ALL PRIORITIES LOWER
THAN THE MASK PRIORITY
ARE LOCKED OUT.

NOTE IN THE RANGE OF
A MASK (I.E BETWE
A CALL MASK and DMASK,
a CALL MASK and DISPAT
NO I/O COMMANDS ARE T
BE USED !!

MASK AND DMASK

Fig. 13


DELAYS
IS THERE A FREE CELL
NO
YES
A=1
CALL ERROR
STORE IN CELL GET "ADDRESS IN PROGRAM" OF PRIMARY CELL.
SEARCH DELAY STACK FOR CELL WITH PRIORITY LOWER THAN OF NEW CELL.
INSERT NEW CELL JUST BEFORE THE CELL FOUND COMPLETE THE CELL RESET POINTERS
DISPATCH
CELL LAY OUT
THREAD
PRIORITY
ADDRESS IN PROGRAM
Q REGISTER
INDICATOR
I REGISTER
CELL
STACK
PRIORITY
MAKE SCHEDULE REQUEST OF NEXTDL.
REQUEST ACCEPTED
NO
RETURN
A=2
CALL ERROR
NEXTDL
CELL
GET Q AND
GET "ADDRESS
GO TO

HERE ALL RELEVANT PARAMETERS OF PROGRAM TO BE SUSPENDED ARE PLACED IN A STACK, TOGETHER WITH THE SUSPENSION ADDRESS, PRIORITY ETC.

NOTE: STACK IS THREADED ON A PRIORITY BASIS AND A FIRST IN FIRST OUT BASIS.

DELAYS AND SDELAY ARE ROUTINES WHICH MAKE IT POSSIBLE TO SUSPEND A PROGRAM TEMPORARELY UNTIL IT IS AGAIN TRIGGERED INTO EXECUTION BY ANOTHER PROGRAM

DELAYS AND SDELAY TO BE ENTERED AFTER A PREVIOUS CALL MASK.

HERE [illegible] EXECUT OF SUSPENDED PROGRA IS CONTINUED

"DELAY SCHEDULE" AND "SCHEDULE DELAY"

Fig. 14

JOB TABLE ENTRIES
(3 MAIN FORMS)

ENTRIES ARE USED BY INITJOB, STATJOB AND ENDJOB ROUTINES

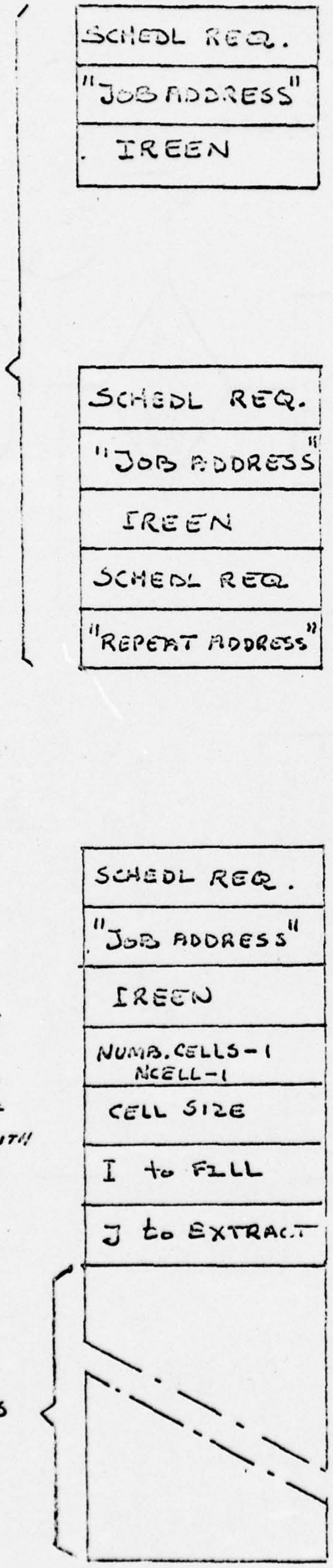

SCHEDL REQ.
"JOB ADDRESS"
IREEN
SCHEDL REQ.
"JOB ADDRESS"
IREEN
SCHEDL REQ
"REPEAT ADDRESS"
SCHEDL REQ.
"JOB ADDRESS"
IREEN
NUMB. CELLS-1
NCELL-1
CELL SIZE
I to FILL
J to EXTRACT
CELLS

TABLE ENTRIES FOR JOBS WITHOUT ARGUMENTS

IF THESE JOBS ARE TO BE CONTROLLED BY A CLOCK THE ENTRY IS PRECEEDED BY TWO LOCATIONS ONE WHICH CONTAINS THE TIME INTERVAL, THE OTHER THE TIME COUNT DOWN.

TABLE ENTRIES FOR JOBS WITH ARGUMENTS IREEN IS NOT ONLY A RE-ENTRY COUNTER, BUT INDICATES ALSO THE NUMBER OF CELLS FILLED WITH ARGUMENTS I AND J are POINTERS TO FILL OR TO EXTRACT THE ARGUMENTS.

THE SECOND FORM OF THE JOB TABLE ENTRY IS FOR THOSE JOBS WHICH WILL NOT RESTART THEMSELF IF IREEN IS NOT EQUAL TO ZERO AT JOB COMPLETION but A SPECIAL (GENERALLY MUCH FASTER) REPEAT ROUTINE.

"JOB ADDRESS" MAY BE THE CORE ADDRESS OF THE JOB, OR THE SYSTEM DIRECTORY ENTRY OF THE JOB

IREEN IS THE RE-ENTRY COUNTER
IT IS INCREASED AT EACH CALL TO INITJOB OR STATJOB

SCHEDL REQ CONTAINS THE SCHEDULE REQUEST COMMAND, THE PRIORITY OF THE JOB OR THE REPEAT JOB IN BITS 3-0

Bits 7-4 IN THE SCHEDL REQ WORD ARE USED TO INDICATE THE FOLLOWING

BIT 4 : 0 IREEN IS SET TO ZERO ON JOB COMLETION

1 IREEN IS DECREASED BY ONE ON JOB COMPLETION

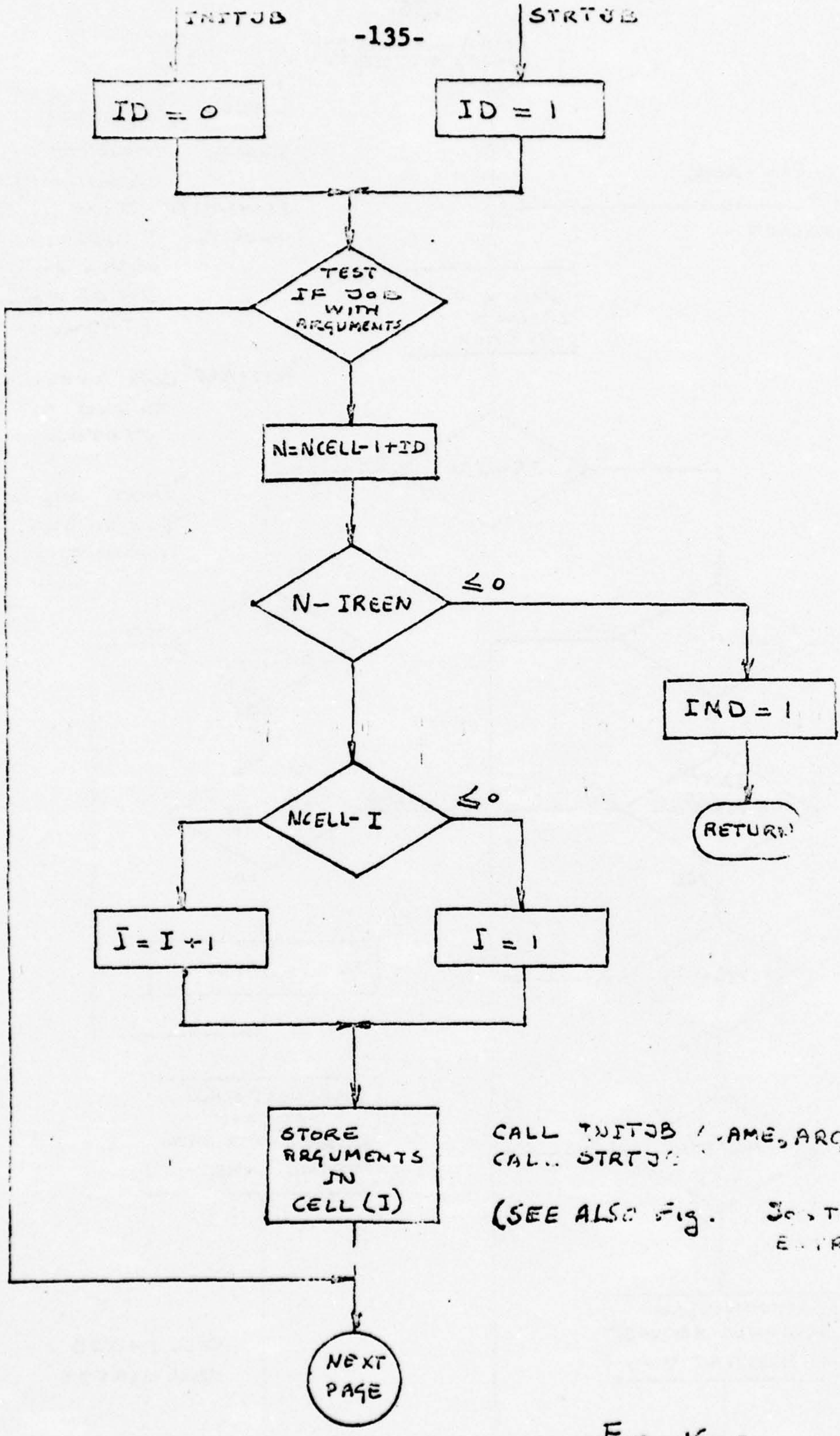
BIT 5 : 0 NODELAY JOB DOES NOT USE DISK

1 DELAY IF DISK MAINTENANCE JOB DOES USE DISK.

BIT 6 : 1 IF ON JOB COMPLETION IREEN IS NOT EQUAL TO ZERO, SCHEDULE THE REPEAT JOB

BIT 7 : 1 JOB IS INITIATED BY CLOCK AT REGULAR INTERVALS THE TABLE ENTRY IS PRECEEDED BY TWO LOCATIONS CONTAINING INTERVAL TIME AND TIME COUNT DOWN

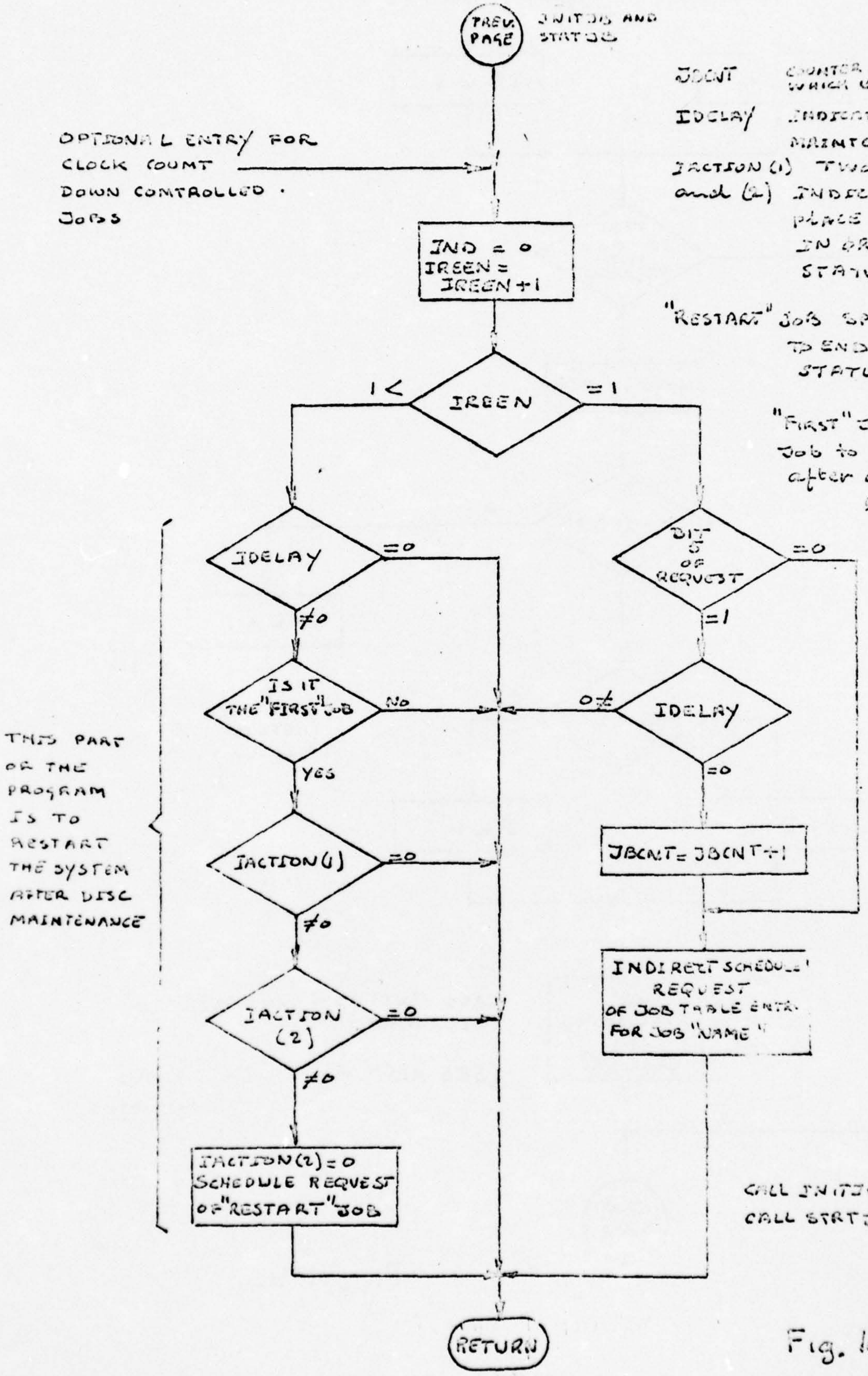
Fig-15


INITJB
STRTJB
ID = 0
ID = 1
TEST IF JOB WITH ARGUMENTS
N=NCELL-1+ID
N- IREEN
≤0
IND = 1
RETURN
NCELL- I
≤0
I = I + 1
I = 1
STORE ARGUMENTS IN CELL (I)
NEXT PAGE

CALL INITJB (NAME, ARG, IND)
CALL STRTJB

(SEE ALSO Fig. 30, TABLES ENTRIES)

Fig. 16 a


PREV. PAGE
INITJB AND STRTJB
OPTIONAL ENTRY FOR CLOCK COUNT DOWN CONTROLLED JOBS
IND = 0
IREEN = IREEN+1
IREEN
1<
=1
IDELAY
=0
≠0
IS IT THE "FIRST" JOB
NO
YES
IACTION(1)
=0
≠0
IACTION (2)
=0
≠0
IACTION(2)=0
SCHEDULE REQUEST OF "RESTART" JOB
BIT 5 OF REQUEST
=0
=1
IDELAY
0≠
=0
JBCNT = JBCNT+1
INDIRECT SCHEDULE REQUEST OF JOB TABLE ENTRY FOR JOB "NAME"
THIS PART OF THE PROGRAM IS TO RESTART THE SYSTEM AFTER DISC MAINTENANCE
RETURN

JBCNT COUNTER OF JOBS IN [illegible] WHICH USE DISK.

IDELAY INDICATOR FOR DISC MAINTENANCE.

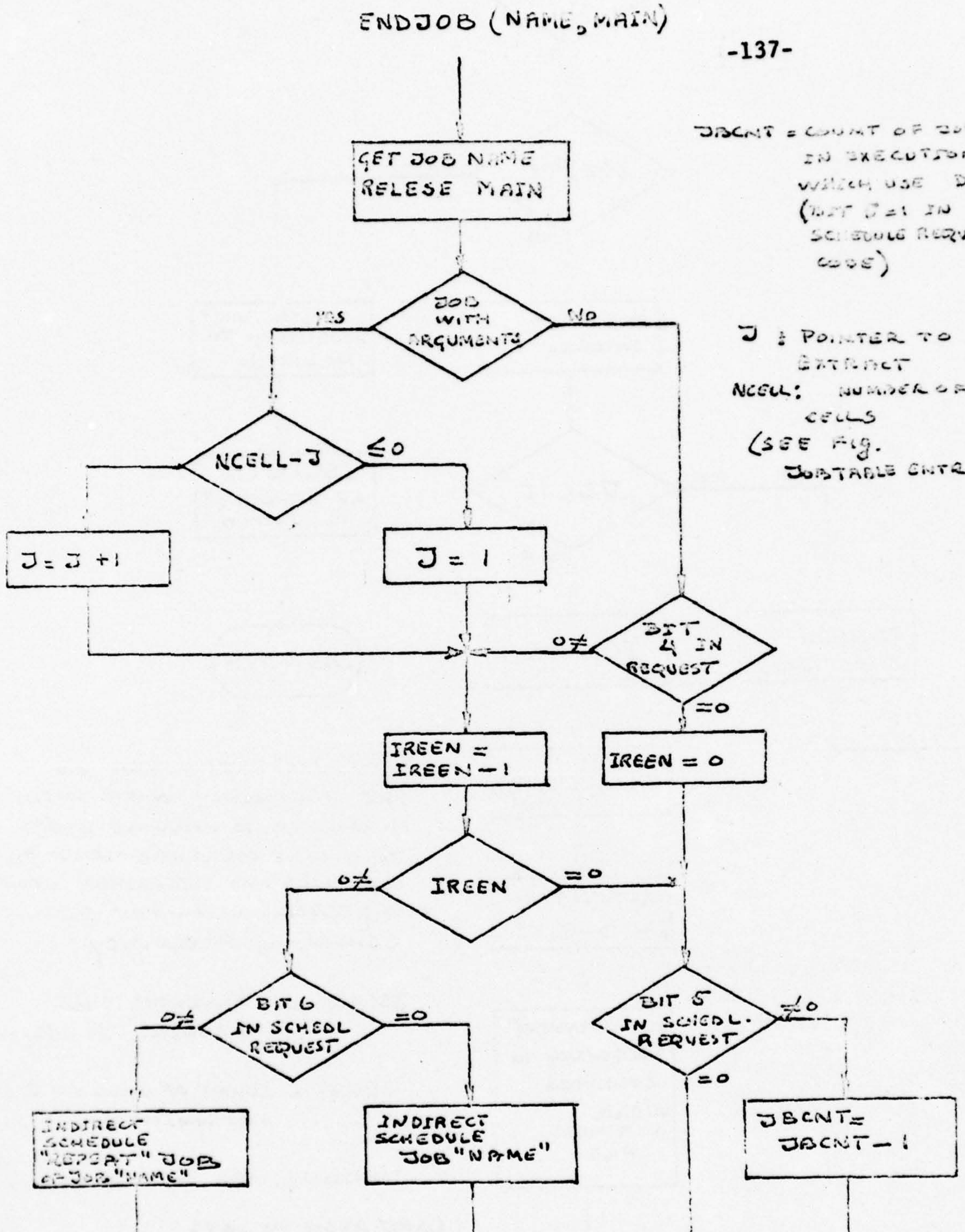
IACTION (1) and (2) TWO ACTION INDICATORS TO place COMPUTER IN OR OUT DELAY STATUS

"RESTART" JOB SPECIAL PROGRAM TO END A DELAY STATUS.

"FIRST" JOB is FIRST Job to be started after a previous delay status.

CALL INITJB (NAME, [illegible] IND)
CALL STRTJB

Fig. 16 b

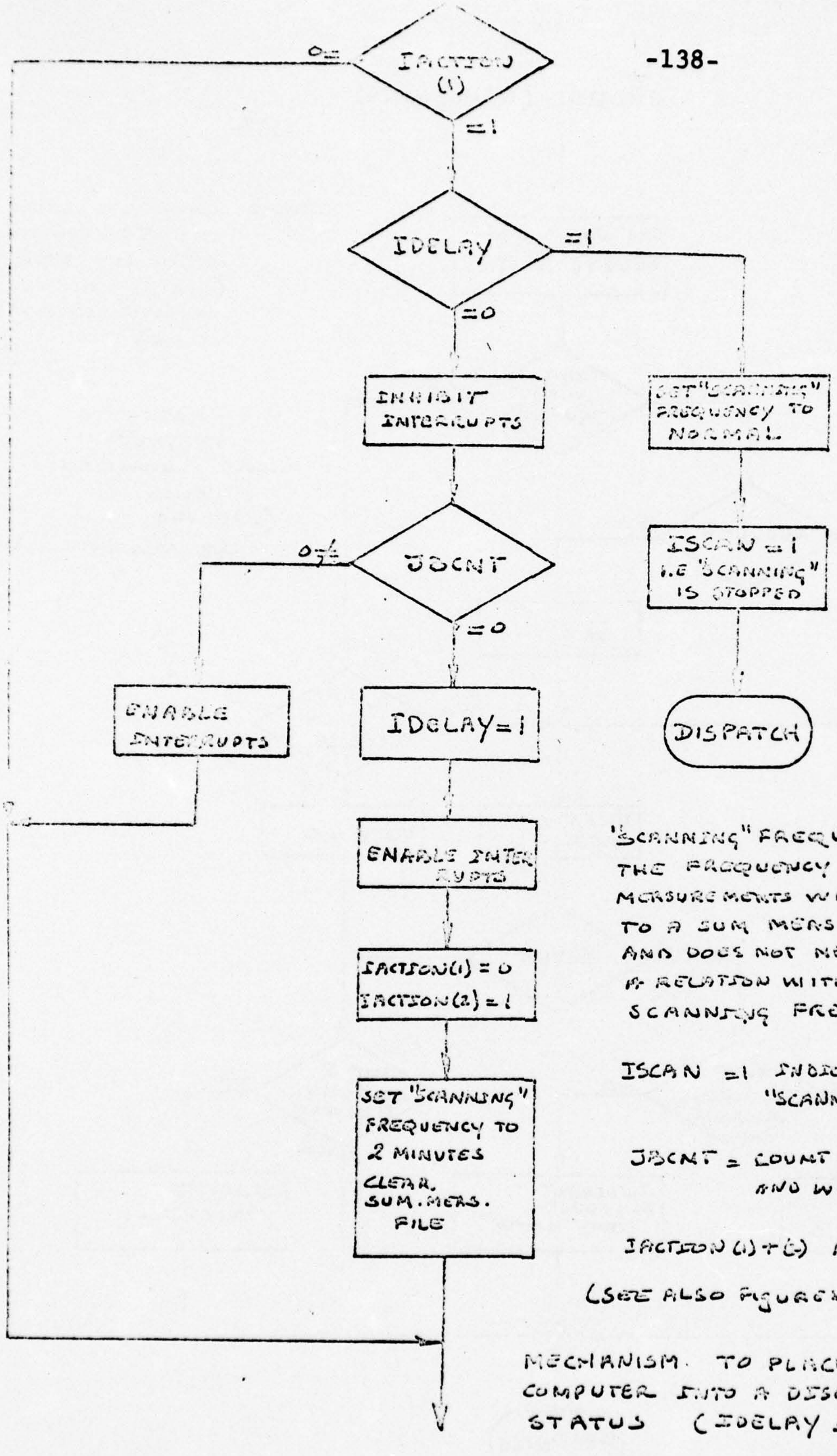

ENDJOB (NAME, MAIN)
GET JOB NAME
RELESE MAIN
JOB WITH ARGUMENTS
YES
NO
NCELL-J
≤0
J = J +1
J = 1
BIT 4 IN REQUEST
0≠
=0
IREEN = IREEN − 1
IREEN = 0
IREEN
0≠
=0
BIT 6 IN SCHEDL REQUEST
0≠
=0
BIT 5 IN SCHEDL. REQUEST
≠0
=0
INDIRECT SCHEDULE "REPEAT" JOB OF JOB "NAME"
INDIRECT SCHEDULE JOB "NAME"
JBCNT= JBCNT − 1
DISPATCH

JBCNT = COUNT OF JOBS IN EXECUTION AND WHICH USE DISK. (BIT 5 =1 IN SCHEDULE REQUEST CODE)

J : POINTER TO EXTRACT

NCELL: NUMBER OF CELLS (SEE Fig. JOBTABLE ENTRIES)

Fig. 17


IACTION (1)
0=
=1
IDELAY
=1
=0
INHIBIT INTERRUPTS
SET "SCANNING" FREQUENCY TO NORMAL
0≠
JBCNT
=0
ISCAN =1 I.E "SCANNING" IS STOPPED
ENABLE INTERRUPTS
IDELAY=1
DISPATCH
ENABLE INTERRUPTS
IACTION(1) = 0
IACTION(2) = 1
SET "SCANNING" FREQUENCY TO 2 MINUTES
CLEAR. SUM.MEAS. FILE

"SCANNING" FREQUENCY IS THE FREQUENCY WITH WHICH MEASUREMENTS WILL BE ADDED TO A SUM MEASUREMENT FILE AND DOES NOT NECESSARY HAVE A RELATION WITH THE REAL SCANNING FREQUENCY

ISCAN =1 INDICATES THAT "SCANNING" IS STOPPED

JBCNT = COUNT OF JOBS IN EXECUTION AND WHICH USE DISC(S)

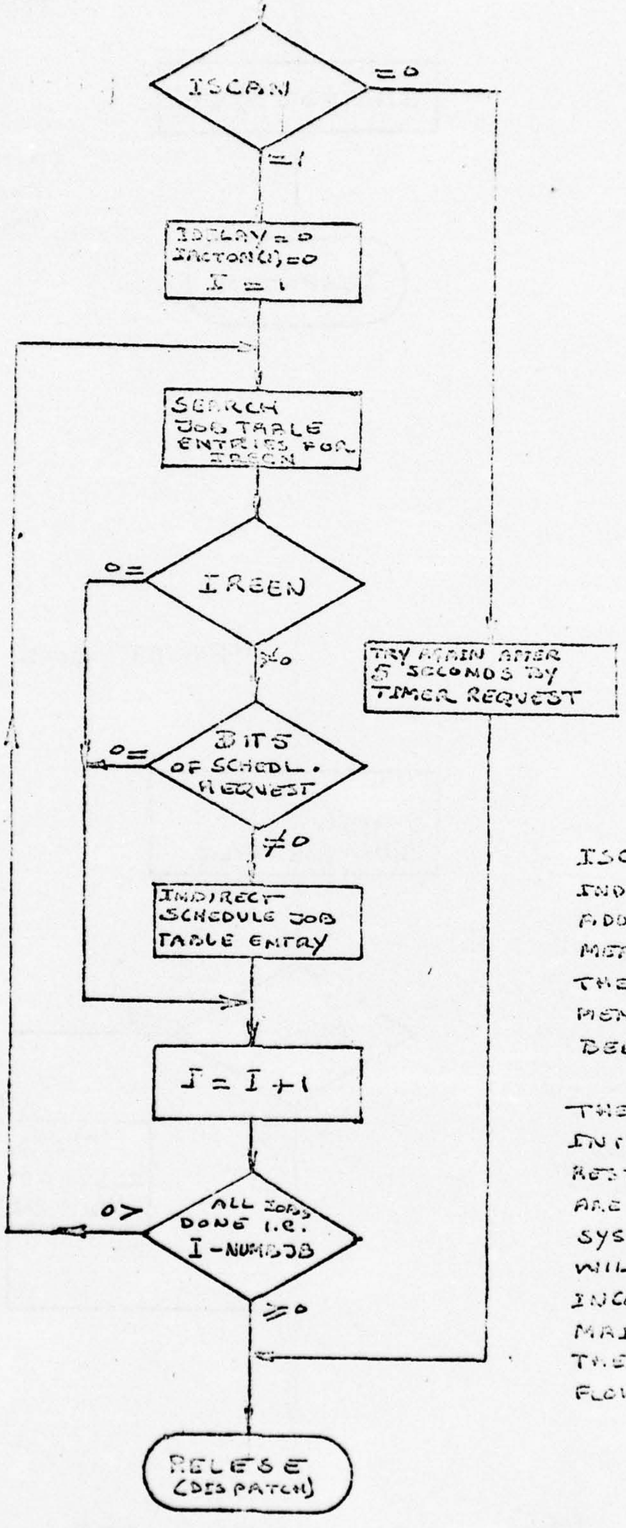
IACTION (1) + (2) ACTION INDICATORS

(SEE ALSO FIGURES)

MECHANISM TO PLACE THE COMPUTER INTO A DISC MAINTENANCE STATUS (IDELAY = 1)

Fig. 13

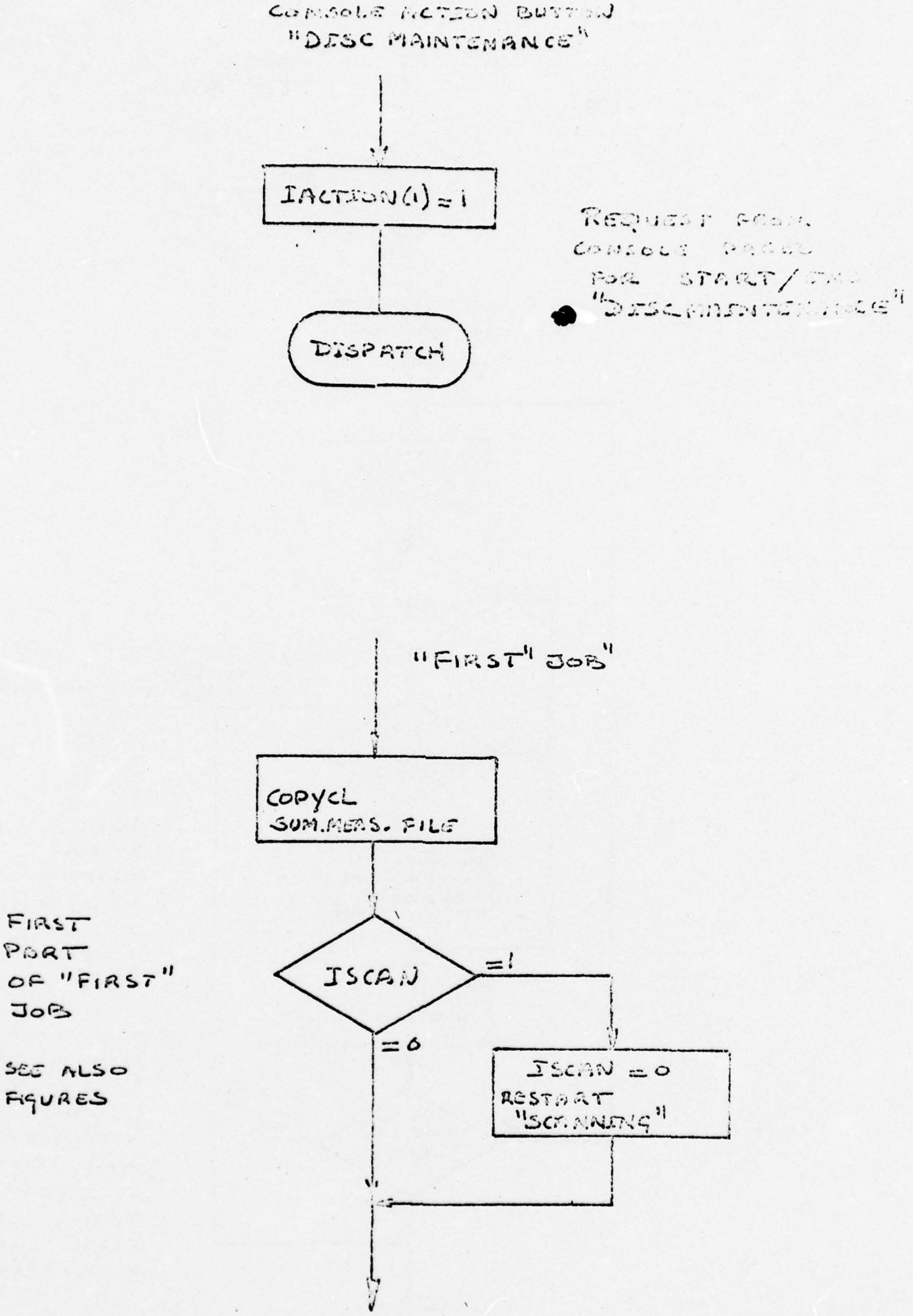
"RESTART" JOB AFTER DISC MAINTENANCE
(SEE ALSO FIG.)


ISCAN
=0
=1
IDELAY=0
IACTON(I)=0
I=1
SEARCH JOB TABLE ENTRIES FOR IRSEN
IREEN
0=
≠0
TRY AGAIN AFTER 5 SECONDS BY TIMER REQUEST
BITS OF SCHEDL. REQUEST
0=
≠0
INDIRECT SCHEDULE JOB TABLE ENTRY
I=I+1
ALL JOBS DONE I.E. I-NUMBJB
0>
≥0
RELESE (DISPATCH)

ISCAN IS AN INDICATOR THAT ADDITION OF MEASUREMENTS TO THE SUM MEASUREMENT FILE HAS BEEN STOPPED

THE ROUTINES TO PUT INTO DELAY AND RESTART THE SYSTEM, ARE MOST LIKELY SYSTEM DEPEND BUT WILL NEVERTHELESS INCORPORATE THE MAIN ITEMS OF THESE PROGRAM FLOW DIAGRAMS.

Fig. 19.


CONSOLE ACTION BUTTON
"DISC MAINTENANCE"
IACTION(1) = 1
DISPATCH
REQUEST FROM
CONSOLE PANEL
FOR START/STOP
"DISC MAINTENANCE"
"FIRST" JOB"
COPYCL
SUM.MEAS. FILE
FIRST
PART
OF "FIRST"
JOB
SEE ALSO
FIGURES
ISCAN
=1
=0
ISCAN = 0
RESTART
"SCANNING"

"DISC MAINTENANCE" AND
"SCANNING" CONTROL

Fig. 20

SECTION V

FORMAT DEFINED LANGUAGES

At the First Workshop on Standardization of Industrial Computer Languages the Problem-Oriented Languages Committee was first organized as the Fill-in-the-Blanks Languages Committee from the popular format for such languages at that time. They recommended a name change to Format Defined Languages Committee and eventually to Problem-Oriented Languages Committee. This first document is their first requirements and definitions report. It was published in the Minutes of that Workshop as Insert VII, pp. 53-57.

FORMAT DEFINED LANGUAGES

A. EXPLANATIONS

1. DIGITAL COMPUTER OUTPUT:
    a. Direct digital control (DDC) is a means of implementing a control function by direct action to the actuator.
    b. Digital setpoint control (DSC) is a means of implementing a control function by setting the set point of an external controller.
2. DIGITAL COMPUTER INTERNAL CALCULATIONS AND CONTROL FUNCTIONS
    a. Regulatory digital control (RDC) uses a large group of control functions short of optimization which may be implemented by cascade adjustment of set points (DSC) or direct action to the actuator (DDC). This includes sequential control as well as the usual closed loop, feedward and other algorithms irrespective of time scale for repetitive actions.
    b. Optimizing control is a control method using special calculating procedures aimed at arriving at optimum set point positions for regulatory loops to meet a specific operating objective.

B. FDL IS RECOMMENDED IN THESE APPLICATION AREAS

1. Data Acquisition by Schedule and Interrupt
2. Data Conditioning
3. Alarm Log and Emergency Response
4. Regulatory Digital Control
5. Support of Man/machine Interface
6. Data Storage and Retrieval
7. Communication with Other Computers

PRECEDING PAGE BLANK-NOT FILMED

C. GENERAL REQUIREMENTS FOR FDL

1. Process Definition
2. On-line Program Expansion and Modification.
3. Should make maximum use of industrial terminology.
4. The language should be self-documenting. The source information after being prepared or listed should be intelligible to the application engineer.
5. Must interface with procedural languages.
6. The application engineer working with FDL should definitely not have to concern himself with bits, bytes or internal number representations, about the real time operating system, about internal interrupt responses, or about routine internal file handling.
7. Sufficient diagnostics should be included to show improper use of the language.
8. Should include methods for the definition of recovery procedures to insure appropriate operation in the event of system failure.
9. The method used to identify and name system equipment should be such that names and addresses need not be defined more than once.
10. Information named within this language should be externally accessible by references to said names.
11. The capability should be provided for an experienced programmer to add new format definitions to the language.
12. Retrieval and documentation of the information after it has been used and modified should be possible.

D. SPECIFIC ITEMS IN FDL

To be specified at subsequent meetings. Example items that may be considered are attached.

EXAMPLES OF SPECIFIC ITEMS TO CONSIDER IN FDL

I. Inputs-Scheduled
  A. Analog
    1. Indentification
    2. Scanning Rate
  B. Pulse
II. Inputs-Interrupt
  A. Analog
  B. Contact Closure
III. Inputs-Generated
  A. Timers
  B. Computed Variables
IV. Alarm and Log
  A. Analog Inputs
  B. Contacts
  C. Computed Variables
V. Conversion of Data & Data Processing
  A. Eng. Units
  B. Aver., Delta, etc.
VI. Control Functions
  A. Build Loops
  B. Build Feed Forward & Interacting Systems
  C. Control Calculations with Constraints
    1. DDC
    2. Set Point (Links to Variety of Calc. Methods)
    3. Sequencing
VII. Control Output
  A. Analog
  B. Contacts
  C. Pulse

VIII. Communications-External
   - A. Oper Console-Pertinent Variable & Control Info. & Adjust
   - B. Engineer
   - C. Report Writing-Typers & Line Printers
   - D. Other Machines
   - E. CRT Displays
   - F. Data Storage and Retrieval

IV. Communications-Internal
   - A. Links to FORTRAN Subr.
   - B. Background Work (Time Share)

X. Misc. Services Routines & Features

E. RECOMMENDATIONS FOR FUTURE ACTION

In order to conform to the adopted goals of the Workshop and to further develop the use of a "Format Defined Language" which specifies parameters and control actions, a committee is to be formed which is directed to do the following:

1. Recommend a generic home for the language(s).
2. Prepare a more complete and detailed set of requirements and features for the language(s).
3. Examine the relationship of these requirements and features to the effectiveness of the existing languages of this type and identify new directions for development.
4. Recommend preparation of a standard language(s) if and when this appears feasible.

SECTION VI

PRELIMINARY REQUIREMENTS
PROBLEM-ORIENTED LANGUAGES

The following three documents present the work of the Problem-Oriented Languages Committee in developing a set of specification requirements for a problem-oriented language. This work is not yet completed since the Committee has concentrated recently on its development of a translator system for these languages (Avenue 2 of the discussion in the Introduction). "The Preliminary Requirements" appeared in the Minutes of the Second Workshop on Standardization of Industrial Computer Languages on pp. 51.63. "The User Criteria for Process Oriented Source Languages" appeared in the Minutes of the Third Workshop on Standardization of Industrial Computer Languages on pp. 73-90. The "Preliminary Recommended Specification and Procedure for a Process Oriented Language", appeared in the Minutes of the Fourth Workshop on the Standardization of Industrial Computer Languages, pp. 88-101.. The "Working Document for the Specification of a Problem Oriented Language" appeared in the Minutes of the Fifth Workshop on Standardization of Industrial Computer Languages on pp. 137-179.

PRECEDING PAGE BLANK-NOT FILMED

PRELIMINARY REQUIREMENTS
PROCESS ORIENTED LANGUAGES

1.0 Introduction

The process oriented language concerns itself with those aspects of industrial computer systems which can be jointly expressed in the following source documents (c.f. Section 2.0):

1. Process scheme and specification
2. Control diagram
3. Logic flow chart.

The process oriented language is a set of rules and symbols which are understandable by humans and mechanies. It expresses information contained in the above three documents in such a form that it can be translated automatically into working computer programs and/or data used by computer programs.

The elements of the language are identifiers and operators. Identifiers (e.g. names, addresses, etc.) designate objects and attributes of objects. Operators (e.g. imperative verbs, arithmetic operators, etc.) designate operation to be performed on objects.

These concepts will be more rigorously defined below.

PRECEDING PAGE BLANK-NOT FILMED

2.0 Source Documents

2.1 Process Scheme and Specification

To describe the process, we consider it necessary to provide:

1. P & I Diagrams
2. Tabular lists of:
   - a) Analog inputs
   - b) Output signals
   - c) Contact signals

The tabulations include such items (but are not limited to) the following:

Analog Inputs

Identification (name, etc.)
Value, alarm limits, etc.
Sensor and Signal, etc.
Sampling
Conversion to engineering units
Notes

Output Signals

Identification
Device and signal
Clamps and alarms
Feedback characteristics

Contact Signals

- Identification
- Device
- Contact rating
- Position (open, closed)
- Sampling (random, synchronous)

2.2 Control Diagram

The control system is defined by a block diagram similar to analog computer or instrument diagrams, using existing standard symbol sets as much as possible. The diagram consists of blocks or special symbols representing computations performed nearly simultaneously, and connections which represent the flow of information from one computation to the other. The calculations include, but are not limited to, two term controllers, dynamic compensators, multipliers, etc. which exist in the analog world or which would exist there if they were practical (dead times). Each calculation would have a number of parts or connections (an exception has been suggested for the equivalent of an analog controller), and might have actions relating to display or other system functions in addition to the control (such as lighting an alarm light).

An alternative to this analog tradition already exists. This is in the "loop" concept, that exists in the many systems, where measurement value, control parameters, and control structure information are conceptualized in blocks together. While this point of view might be accommodated in the approach we are taking here, we feel that a major reason for not doing so is related to the need for effective operator interface to the control system. The result is a specification for digital control techniques oriented about the analog tradition. We believe that operator interfacing is sufficiently important and detailed to require separate specifications and we question whether, as a standard, tieing control and display together can be made sufficiently flexible and general to satisfy most users.

From the original analog point of view a control system is defined by a block diagram and a list of parameters. For purposes of discussion the control system is described in terms of the following entities:

- A control system is an entity with a name and a series of control elements.
- A control element is the digital equivalent of an analog computing block. It is an entity with a name, a type (controller, multiplier, calculation, etc.) and several variables associated with inputs, outputs, and parameters which may be used as an input to another calculation.
- A variable is an entity having name, arithmetic type, engineering units, data entry information (whether the value is calculated, entered manually or entered from the supervisor).
- A macro element is an element whose computation is defined in terms of other elements (effectively in a diagram) but which is treated in other respects like a control element. We assume the capability to define macro elements out of other elements or as computational procedures.

By the way of illustrating these concepts, a tabular representation of a series of control elements is given in Figure 1. It is assumed that the element variables are all indicated by symbolic references and that any undefined variables can later be specified by a table of the form shown in Figure 2. The variable references are grouped in Figure 1 in order as outputs, inputs and parameters. There is a single main output which is assigned the name of the element unless otherwise specified in the table (although there are calculated values to be stored with the element which are not explicit in the table such as the integration in a PID). The purpose of the table is to show the nature of the control system specification. While this might appear to provide a basis for a fill in the blanks implementation, it is not our intent to describe the form of the language.

It will be noticed that in Figure 1 connections are made by referring to common variables or to the output of a preceding variable. As a matter of principle, we would prefer free bi-directional interconnection between all variables (including any stored values) intermediate to final results. If the name of the main output was used as the name of the main variable, subscripts could be used to access secondary variables.

Definition of entries in Figure 1:

FRC104 is a PID controller with 1 sec sample time

V103 is its valve output

F101 is its measurement (a flow)

SP104 its setpoint

Initial Status: Auto-manual, positional - incremental, increase-decrease, etc.

KP104 Proportional gain

KI104 Integral gain

KD104 Derivative gain

HL104 Hi limits to valve travel

LL104 Lo "

CALC No.4 is a multiplier for

F85, a flow measurement and

P85, a pressure measurement giving as a result a compensated flow.

CALC No. 5 is a sample and hold element which picks the peaks of A33, an analyzer, under logical control of CMP10, a comparator.

The variables are specified in terms of their real or logical character, their conversion properties, and their data entry (or operator interface) propertie
The assignment of a name and type to a control element is a convenient means of tabulation, and does not imply any particular method of language implementation.

F I G U R E 1 CONTROL ELEMENTS

NAME	SAMPLE TIME	TYPE	OUTPUT	INPUTS		Initial Status	PARAMETERS				
FRC104	1	PID	V103	F101	SP104		KP104	KI104	KD104	HL104	LL104
CALC#4	1	MUL		F85	P85						
CALC#5	1	S&H		A33	CMP10						

F I G U R E 2 VARIABLES

NAME	ARITH. TYPE	ENG. UNITS	ENTRY FORM M,S,C,	INITIAL VALUE	
KP104	Real		M		
FL104	Logical		M		

2.3 Logic Flow Charts

It is proposed that the process be delineated by the flow charts. The flow charts completely describe the process logic and operations to be performed during normal operation, and also under exceptional or emergency conditions.

Basically, the flow charts describe how the equipment and devices specified on the instrumentation and control diagram are manipulated to accomplish the required actions.

It is required that a system have a provision to initiate or terminate any procedure in response to:

a) a random external event

b) a request by another procedure

c) an abnormal condition detected within a procedure during execution.

Conditions for initialization and termination are described on each logic flow chart and thereby describe the other procedures or events in the system.

The logic flow chart consists of a group of operations represented by suitable standard symbols. (Note: Each symbol and its contents correspond to a single language statement.) The symbols are connected with lines and labels, so that the process operation is completely described.

It is noted that six (6) standard symbols are sufficient to describe any process sequence. These are the Query, Action, Message, Connection Label, Initiate, and Terminate symbols.

The Query symbol is represented by —⟨expr⟩— and has the following meaning:

If expr = TRUE go to right, otherwise go down. The "expr" may be any relational or logical expression. For example, "expr may be:

FT1 $\geqslant$ 40, AND FTI $\leqslant$ 80; or (DI1 AND DI2) OR DI3; where FT1 is a real value and DI1, 2, 3 are digital inputs.

The action symbol is represented by —┤ ├— and means any unconditional operation by the system. Examples of such unconditional operations are:

a) Evaluate an arithmetic expression
b) Operate an external analog or digital hardware device
c) Read and save the value of an analog input device
d) Read and save the condition of a digital input device
e) Initiate a conditional or unconditional time delay
f) Change the value of attributes describing an analog input
g) Change the value of attributes describing a DDC algorithm
h) Execute another procedure

The Message symbol is represented by —▽n— and means the unconditional output of the message referenced by n.

The Connecting Label symbol is ⓝ, where n is a numeric, alpha, or alpha numeric string and in general refers to another connecting label containing n in the flow chart of procedure.

The Initiate and Terminate symbols are ▷— and —◁ respectively, and represent the beginning and ending of a procedure.

3.0 Man-Machine Communication

Man-machine communication utilizes devices such as typers, printers, keyboards, consoles, and CRT's.

The scope of the Process Oriented Language includes the content of the communication between man and machine, but excludes the formats of the messages used in communication.

4.0 Classification of Objects

4.1 Devices

The term "device" includes such things as process actuators and sensors, control loops, or even complete process units.

A simple device is one which is not composed of other devices.

A composite device is one which is composed of other devices, either simple or composite.

4.1.1 Attributes of Devices

In general, devices have the following attributes:

1. Description (name)
2. Components (parts list)
3. Variables

As a specific example, a simple external device such as a valve, has the following attributes:

1. Description (name)
2. I/O elements (latching relay)
3. I/O variable (eg. actual and desired state)

4.2 Variables

A variable is any arbitrary data structure.

4.2.1 Attributes of Variables

Example: The following attributes are representative of those associated with an analog input variable:

Name
Value and initial value
Process Unit ID
Change criteria
Signal properties
Engineering unit conversion
Filtering
Sample rates
Alarm limits
Reasonableness
Rate of Change limits
Deadband
Priority of point
Data type
Synchronization
Data compression
Demand read
Miscellaneous

4.3 Procedures

Procedures such as programs and sub-programs are within the scope of this language.

5.0 Classification of Operations

5.1 Arithmetic and Relational

$+, -, *, /, \uparrow, >, <, \geqslant, \leqslant, \neq, =$

Function Library

5.2 Logical

And, or, not, exclusive or

5.3 String

Move, edit

5.4 Transfer Control

5.4.1 Unconditional

Do, until, while, go to, delay

5.4.2 Conditional

If, else, when, wait until

5.5 Procedure Control

Start program, stop program, call subroutine, return, exit.

6.0 Language Definition

6.1 Longhand Form

In the longhand form, some or all of the elements of the language are separated by symbols called delimiters. In this form, the language resembles a typical procedural language such as FORTRAN or AUTRAN.

6.2 Shorthand Form

The elements of the language occupy fixed relative character positions in the input stream and are not separated by delimiters. In this form the language resembles a typical fill-in-the-blanks language such as PROSPRO or BICEPS.

6.3 Publication Conventions

The process of preparing this specification and subsequent user manuals for printing requires a form of expression which makes the reading of printed text easier. It is therefore necessary to define conventions to be used when describing the language in publications.

USER CRITERIA FOR PROCESS-ORIENTED SOURCE LANGUAGE

User criteria are the attributes of the Program-oriented Language (POL) which enable the user to define his process to the computer system. The computer system is represented by (Figure V-1) a group of functional processors interconnecting through a central processor. The user communicates his process requirements to these processors by a combination of statements representable in Bachus Normal form consisting of specification, sequences, procedures and conditionals.

Sequences can be implemented on any of the processors. They consist of a string of sentences initiated by a conditional which is implemented through the Central Processor (CP). The physical process which can be a batch, or continuous process is implemented with sequences in the respective functional processor and initiated at system start up. The sequence describing the batch process are grouped in a separate Batch Processor.

The attributes of the following processors are described below.

1. Analog Input
2. Analog Output
3. Digital Input
4. Digital Output
5. DDC
6. Man-Machine Communication
7. Computation, Arithmetic and Logic
8. File Builder
9. Batch Processor
10. Cold Start and Maintenance

Those attributes of the functional processor which are static information are characterized in the specification state-

PRECEDING PAGE BLANK-NOT FILMED

ments and the dynamic operations which act on the specifications are characterized in the procedure statements.

CHARACTERISTICS COMMON TO ALL MACHINES

1. A program memory and a date memory (but common to all machines)
2. A program counter (all machine programming languages are assumed to be sequential.)
3. A repetition stack in which is stored a list of frequency code-address pairs indicating that, when free, the machine will jump to any address from the stack in its program whose time-for-next repetition condition has been met.
4. Has a terminate command TMT which releases machine to process next request.
5. Has a repeat command:

   RPT freq init.

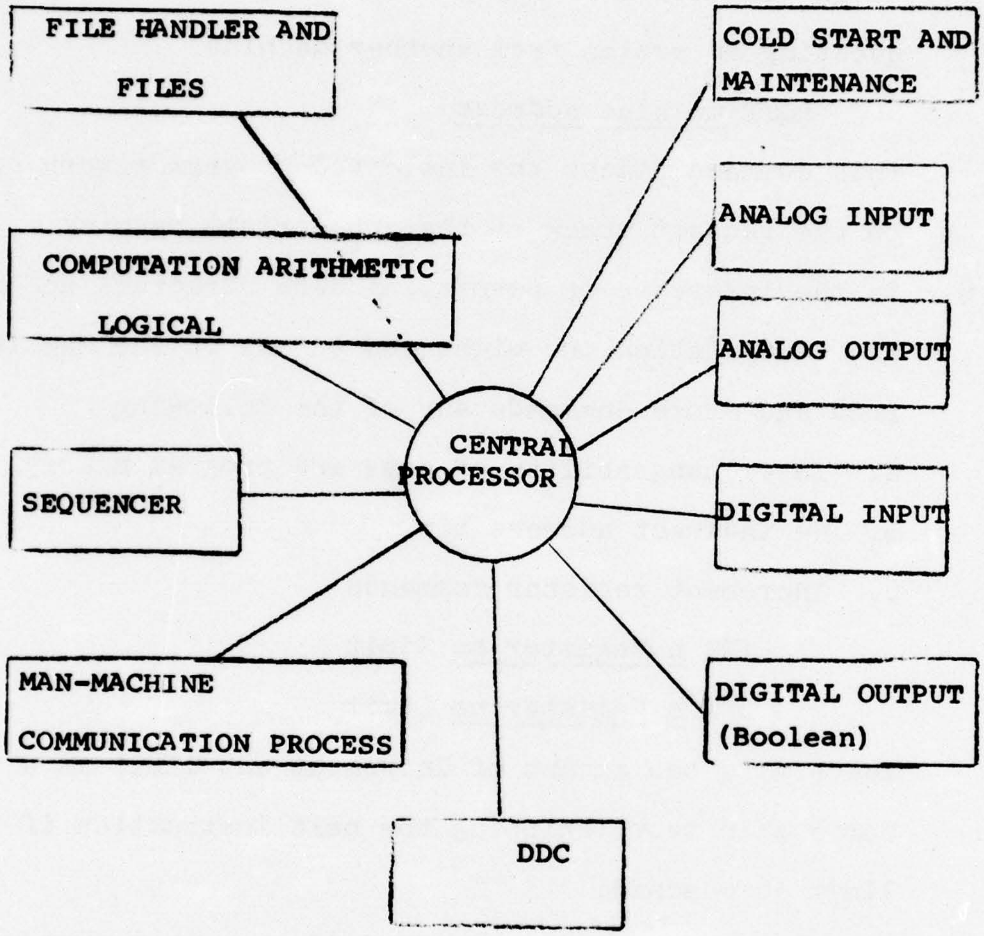
   which causes the frequency code to be placed on the repetition stack along with the address after that of the RPT command. Also has a release command: RLS address, which deletes any pair with the given address from the repetition stack.
6. A Time marker command:

   TM address

   whose function is to increment a time marker (see coordinating machine) and to release processing from present program if the time marker count is completed. Also sends

FIGURE V-1

SYSTEM VIEW


FILE HANDLER AND FILES
COLD START AND MAINTENANCE
ANALOG INPUT
COMPUTATION ARITHMETIC LOGICAL
ANALOG OUTPUT
CENTRAL PROCESSOR
SEQUENCER
DIGITAL INPUT
MAN-MACHINE COMMUNICATION PROCESS
DIGITAL OUTPUT (Boolean)
DDC

$$\text{Initiation} \quad \left\{ \begin{array}{l} \text{T/S} \\ \text{Batch (Priority)} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{External} \\ \text{Time (Duration,clock)} \\ \text{Internal} \end{array} \right\}$$

a return address to the coordinating machine-time marker stack.

7. A machine request stack containing addresses to be jumped to on completion of any present program and a command requesting an action from another machine.

   RQST machine address

   This command places the indicated program memory address on the request stack of the appropriate machine.

8. In the interests of permitting data structure definition and manipulation one might add to any of the machines with Load and Store commands any of the following

   a. Interchangeability of data and program memory

   b. An indirect address bit

   c. Increment register commands

      INR n Register no limit

      DCT n Register no limit

   where n is the amount of increment and limit is a value for a skip test, skipping the next instruction if the limit is reached.

   d. With reservation we might add a minimum number (5 or or 10 at most) arithmetic commands and register (add complement, and and/or) and skip tests.

COORDINATING MACHINE

1. Has a time marker stack (Dijkstra semaphore) and a command, STM n address, (set time marker stored at address to count n completed conditions before release). The

time marker stack contains marker address - machine-program address triplets such that, when free, any machine with an entry in the stack corresponding to a completed time marker will be returned to the program address. This permits a uniform way of synchronizing actions without requiring any assumptions about who triggers what program.

DDC MACHINE

1. Has a set of standard numbered registers of sufficient number. Each register would allow standard data information storage but would allow an "undefined" code as well.
2. Has a load register command:

   LD Register number address

3. Has a store register command:

   ST Register number address

4. Has a "de-defining" command, DED, which puts all registers in an undefined state.
5. Has an algorithm command:

   ALG type

   whose function is to act on the entire set of registers as a block and apply to these an action appropriate to the type code. There would be standard algorithm types such as:

   Controller
   Lead Lag
   Square Root, Adder, Diffencer, etc.

There might also be a "higher level language" type whose only function would be to transmit the data to the "higher level language" machine along with a request for action and a return address. The action would then terminate.

As an example of the function of this machine consider the controller algorithm. A list of data normally dealt with in a controller might be as follows:

1. Error
2. Auto manual status
3. Manual input
4. Valve analog feedback
5. Valve too low digital signal
6. Valve too high digital signal
7. Feed forward
8. Proportional
9. Reset
10. Derivative
11. Derivative filter gain
12. Valve
13. Δ Valve
14. Derivative Data
15. Integral Data
16. Mode Code

Certain (or all) of these data forms might be optional in which case the algorithm would recognize any undefined registers and act accordingly (Such options might also be defined explicitly in a mode code) For instance, although the controller in its most general coding is three term, we might compile a two term controller driving a valve into code as follows:

```
RPT     1 sec           0
DED
LD      1               ERROR
LD      2               AM
LD      3               MAN
LD      8               PROP
LD      9               RESET
ALG     CONTROLLER
STA    12               VALVE
RQST   AO               OUTVALVE
TMT
```

This last command is a request to the analog output machine to go to the address of a routine that outputs the valve.

ANALOG INPUT

1. Has an address, a condition register, and a device address register plus several data registers with loading and storage commands.

    LD Register no. address

    ST Register no. address

2. Has increment and decrement register commands

    INR n register no limit

    DCR n register no limit

3. Has input command

    RAS type

    (Read analog sensor) where the type code would cover standard device types such as current (or voltage) out, millivolts out, pulse width out, etc. and where the address register is where data is stored; the device register indicates which point data is input from.

4. Has skip commands

   SOF n (skip on fault) where n is the bit in the condition register

5. Has filter and convert commands acting on data in registers and at the address in the address register

   FLT type

   CONV type

ANALOG OUTPUT

Same as analog input except for items 3 and 5.

3. Has output command

   OAV type (Output Analog Value)

5. Has conversion command

   CONV type

DIGITAL INPUT

Same as analog input except for items 3,5 and it has a bit field register for defining the bit read into.

3. RDS type

   (Read digital signal)

5. Masking commands and extra registers

   CMR register no (compliment register)

   AND register no address

   OR register no address

6. Skip commands:

SKBS n address(Skip if bit n of the word in the address is set).

DIGITAL OUTPUT

Same as above with

3. ODV type Output digital value

HIGHER LEVEL LANGUAGE MACHINE

Same type machine with higher level language available.

CONSOLE MACHINE

1. The machine would have a keyboard register or registers permanently containing a code defining the "ored" state of itself and the most recent keyboard actions on a bit by bit or field by field basis.
2. It would have a data register with Load and Store commands.
3. AND, OR, and bit skip test commands as in Digital Input
4. Display commands
    a. DSP device

    Transmits code in data register to console device

    b. DSPC char x y

    Positions are given character at position x, y on the scope

    c. DSPVC mode x_1 y_1 x_2 y_2

    Positions a vector of given mode (solid, dotted, etc.) from point x_1 y_1 to point x_2 y_2 on scope.

SOME DEFINITIONS

Backus Notation

A notation used to describe the syntax of languages. The notation is defined below.

1.0 Metalinguistic Variable

< p > A sequence of characters enclosed by the bracket termed a metalinguistic variable.

The values of a metalinguistic variable are sequences of symbols.

2.0 Metalinguistic Connectives

(1) Explicit Connectives

: := "IS defined as"

Example: a ::= b reads as " a is defined to mean b "mean b "

"inclusive or"

Example: a / b reads as " a or b "

(2) Implicit Connective

The absence of an explicit connective between two metalinguistic variables implies the presence of the connective "and".

Example: a b reads as - " a and b ".

3.0 Metalinguistic Constants

Any symbol which is not a variable or a connective is termed a constant and denotes itself.

Example: The string of symbols JOE is a metalinguistic constant. A permissible synomum of metalinguistic constant is literal. Also, literals may be optionally underlined for easier reading.

4.0 Syntax Equations

A syntax equation is the definition of a metalinguistic variable.

Examples:

(1) <letter> : := A/B/..../ i.e.

"The variable letter is defined as one character of the alphabet".

(2) <digit> : := 0/1/2/3/4/5.../9 i.e.

"A digit is one of the characters 0,1,2, etc."

(3) <alphanumeric character> : := <letter> / <digit>

"An alphanumeric character is a letter or a digit".

(4) <integer> : := <digit> / <digit> <integer>

"An integer is a digit, or digit immediately followed followed by an integer".

5.0 Backus Normal Form of Language Definitions

The set of syntax equations which completely define the syntactic rules of a language.

FORM OF A PROGRAM AND SENTENCE

<PROGRAM UNIT> = <TITLE> <PROGRAM BODY> <END>
<TITLE> = <HEADER> <LIST>
<HEADER> = SPECIFICATION/PROGRAM/SUBROUTINE/
REAL FUNCTION/INTEGER FUNCTION/LOGICAL FUNCTION
<PROGRAM BODY> = <SENTENCE>/
<SENTENCE> <PROGRAM BODY>

<SENTENCE> = <IMPLICIT SENTENCE> /
<EXPLICIT SENTENCE>

<EXPLICIT SENTENCE> = <SENTENCE ID> <DELIMITER> <IMPLICIT SENTENCE>

<IMPLICIT SENTENCE> = <CLAUSE LIST> <SENTENCE TERMINATOR>

<CLAUSE LIST> = <CLAUSE> / <CLAUSE> <DELIMITER> <CLAUSE LIST>

<CLAUSE> = <SPECIFICATION CLAUSE> /<DECLARATION CLAUSE> /
<EXECUTABLE CLAUSE>

<END> = END <SENTENCE TERMINATOR>

Underfined Productions:

SENTENCE TERMINATOR

DELIMITER

DECLARATION CLAUSE

EXECUTABLE CLAUSE

SENTENCE ID

FORM OF A SPECIFICATION CLAUSE

<SPECIFICATION CLAUSE> = <IMPLICIT SPECIFICATION CLAUSE> /
<EXPLICIT SPECIFICATION CLAUSE>

<EXPLICIT SPECIFICATION CLAUSE> = <CLAUSE ID> <DELIMITER>
<IMPLICIT SPECIFICATION CLAUSE>

<IMPLICIT SPECIFICATION CLAUSE> = <OBJECT ID> <DELIMITER>
<DATA LIST>

<DATA LIST> = <DATA ITEM> /<DATA ITEM> <DELIMITER> <DATA LIST>

<DATA ITEM> = <LIST> / <DATA TYPE ID> <DELIMITER> <LIST>

Undefined Productions:

CLAUSE ID

OBJECT ID

DATA TYPE ID

DELIMITER

FORM OF LIST AND NAME

<LIST> = <LIST OF NAMES> / <LIST OF VALUES> /
<LIST OF NAMES> <DELIMITER> <LIST> /
<LIST OF VALUES> <DELIMITER> <LIST> /

<LIST OF NAMES> = <NAME> / <NAME> <DELIMITER> <LIST OF NAMES>

<NAME> = <VARIABLE NAME> / <DEVICE NAME> / <PROGRAM NAME>

<LIST OF VALUES> = <NUMERIC VARIABLE> /
<LOGICAL VARIABLE> /
<NUMERIC VARIABLE> <DELIMITER> <LIST OF VALUES> /
<LOGICAL VARIABLE> <DELIMITER> <LIST OF VALUES>.

Undefined Productions

DELIMITER

VARIABLE NAME

DEVICE NAME

PROGRAM NAME

LOGICAL VARIABLE

FORM OF VARIABLE AND ATTRIBUTE

```
<VARIABLE> = <ATTRIBUTE> /<ATTRIBUTE> <END OF FIELD> <VARIABL

<ATTRIBUTE> = <NUMERIC VARIABLE> /
              <LOGICAL VARIABLE> /
              <CHARACTER VARIABLE> /
              <NUMERIC VARIABLE> <END OF FIELD> <VARIABLE> /
              <LOGICAL VARIABLE> <END OF FIELD> <VARIABLE> /
              <CHARACTER VARIABLE> <END OF FIELD> <VARIABLE>

<NUMERIC VARIABLE> = <REAL> / <INTEGER>
```

Undefined Productions:

REAL

INTEGER

LOGICAL VARIABLE

END OF FIELD

CHARACTER VARIABLE

EXAMPLE OF DESIGN CRITERIA FOR FUNCTIONAL PROCESSORS

Specifications (sensor characteristics, and parameters)

1. Procedure to input e.g., A/D multiplexer, address, voltage scaling information etc.
2. Validation
3. Conversion parameters
4. Filter parameters
5. Alarm limits

Procedures (sensor operation and data processing)

1. Rate of scan
2. Base time
3. Order
4. Action of completion
5. Storage
6. Type of scan
7. Stop scan
8. Options e.g., filter, alarm checking, etc.

DIGITAL INPUT (Boolian)

Specifications

1. Input Address (Control Word)
2. Word size, bit position
3. Initial Conditions or Normal State
4. Msgs. (Normal, Abnormal, Changes)

Procedure

1. Rate
2. Base Time
3. Actions
4. Logical expressions of interrelations with other digital input

DIGITAL OUTPUT (Boolian)

Specifications

1. Output Address
2. Word size, bit position
3. Duration
4. Msgs. (such as confirmations)

Procedure

1. Rate
2. Base Time

DDC

Specifications

1. Type
2. Input ID's
3. Output ID's
4. Parameters
5. Alarms (No control alarm, etc.)
6. Msgs (Including displays, etc.)
7. Set point

Procedures

1. Rate
2. Base Time
3. Set point changes

MAN MACHINE COMMUNICATION - OUTPUT

Specifications

1. Device Name (Tel-type, CRT, Mag Tape)

Procedure

1. File Name
2. File Description (e.g. Floating pt. etc.)
3. Format (including graphs)

FILE BUILDER

Specifications

1. Variable Value Files

   2. Process Constants
   3. Message files

Procedure

   1. Build
   2. Delete
   3. Modify

GLOSSARY

Semaphore - A counter mechanism for coordinating parallel processes originated by Dijkstra and occurring in British RTL. The idea behind it is that one may initialize a counter so that as each of several processes arrives at a desired point, it increments the counter and hangs up until the count is completed and all processes have at a condition of coordination.

Virtual - Adjective applied to a hardware-software system which is designed to look like or simulate an equivalent plausible hardware system.

Virtual Memory - A mass storage filing point of view which permits programs to operate as if they had unlimited core (up to the amount of mass storage available) and machine instruction set.

Virtual Machine- An abstract machine (which could be simulated in another machine) used to define the meaning of more elaborate languages or to facilitate inter-machine transportability of programming languages. Generally it would be designed to be simply simulated and unambiguously interpreted, whereas the more elaborate languages would be more difficult to compile directly. A general compiler may then be written into the virtual machine.

The following definitions will be prepared by the Committee:

Definitions:

Analog Input

Analog Output

Digital Input
(Contact closer)

Digital Output

Functional Processor

Central Processor

Batch Process

Continuous Process

Sequences

Conditionals

Language

Procedure

Object

Variable

Data

Operators

Devices (simple composited)

PRELIMINARY RECOMMENDED SPECIFICATION AND PROCEDURE FOR A PROCESS ORIENTED LANGUAGE

1. VARIABLES

1.1 Process Variable Input

Specifications

(1)	Signal Type	;	Analog Input Pulse Input Code Input
(2)	Procedure to Input	;	Device Address Terminal Address Processing Sequence (Scan Block)
(3)	Validation	;	Validity Check Required Action/Message
(4)	Conversion Parameters	;	Compensation Conversion Equations Coefficients
(5)	Filtering	;	Filter Type Filtering Factors
(6)	Limit Check	;	Instrument Limit Process Variable Limit Rate of Change Limit Set Point Limit No. of Repeat reads before declaring fault Deadband to Action Required Action/Message
(7)	Information	;	Variable Description Eng. Unit Variable Name

Procedures

(1) Read Mode: Random, Sequential
(2) Action on Completion
(3) Rate of update
(4) Base Time of update
(5) Accumulation of past results
(6) Average and Min. Max.
(7) Storage
(8) Option ; Error Condition Keep
Read Stop
Recovery Action
Replace Action of Variable

1.2 Process Variable Output

Specifications

(1) Signal Type ; Pulse Train
Pulse Width
Code
Analog Output

(2) Procedure to Output ; Device Address
Terminal Address
Processing Sequence

(3) Validation ; Illegal
Required Action/Message

(4) Conversion Parameters ; Absolute/Incremental
Code Conversion
Encode

(5) Limit Check ; Instrument Limit
Move Limit
Rate of Change Limit
Feed Back Input Limit
Required Action/Message

(6) Information ; Variable Description
Eng. Unit
Variable Name

Procedures

(1) Order of the output
(2) Action on Completion
(3) Rate of output
(4) Base Time of output
(5) Computer Override ; Automatic
Manual
Hold

(6) Option ; Error Condition

1.3 Logical (digital) inputs

Specifications

(1) Input Address ; Device Address
Terminal Address
Bit Position

(2) Validation ; Validity
Required Action/Message

(3) Conversion Parameter ; Mask for Bits Selection
Logical Expression

(4) Initial Condition or Normal State

(5) Alarm Condition ; Change Status
Abnormal
Required Action/Message

(6) Information ; Status Description
Bit Name

Procedures

(1)	Read Mode (random, sequaential)		
(2)	Action on Completion		
(3)	Rate of update		
(4)	Base Time of update		
(5)	Detection of State Change		
(6)	Storage		
(7)	Option	;	Error Condition Keep Read Stop Recovery Action Replace Action

1.4 Logical (digital) output

Specifications

(1)	Signal Type	;	Momentary Latch Pulse
(2)	Output Address	;	Device Address Terminal Address Bit Position
(3)	Validation	;	Illegal Required Action/Message
(4)	Conversion Parameters	;	Mask for Bit Selection Logical Expression
(5)	Normal State or Initial Condition		
(6)	Alarm Condition	;	Abnormal Feed Back Check Required Action/Message
(7)	Information	;	Status Description Bit Name

Procedures

(1) Order
(2) Action on Completion
(3) Rate of Output
(4) Base Time of Output
(5) Duration of Pulse
(6) Computer Override ; Automatic
    Manual
    Hold
(7) Option ; Error Condition Keep
    Feed Back Input Action
    Output Halt
    Recovery Action
    Replace Action

1.5 User's Interrupt

Specifications

(1) Interrupt Type ; Hardware, Software
(2) Interrupt Address ; Level
    Position
(3) Conversion Parameter ; Mask for Receiving
    Logical Expression
(4) Alarm Condition ; Normal
    Abnormal
(5) Information ; Status Description
    Point Name

Procedures

(1) Interrupt Receive ; Mask or Unmask
(2) Branch Vector ; Level Assignment
    Branch Address

(3) Action on Completion

(4) Save Status

1.6 Intermediate Variable

Specifications

(1) Variable Address

(2) Calculation ; Calc. Equation
    Independent Variables
    Coefficients

(3) Limit Check ; Variable Limit
    Rate of Change Limit
    Deadband to Action
    Required Action/Message

(4) Information ; Variable Description
    Eng. Unit
    Variable Name

Procedures

(1) Order of Update

(2) Action on Completion

(3) Rate of Update

(4) Base Time of Update

(5) Storage

(6) Option ; Reset

2. DDC

Specifications

(1) Control Type

(2) Process Variable Input ID

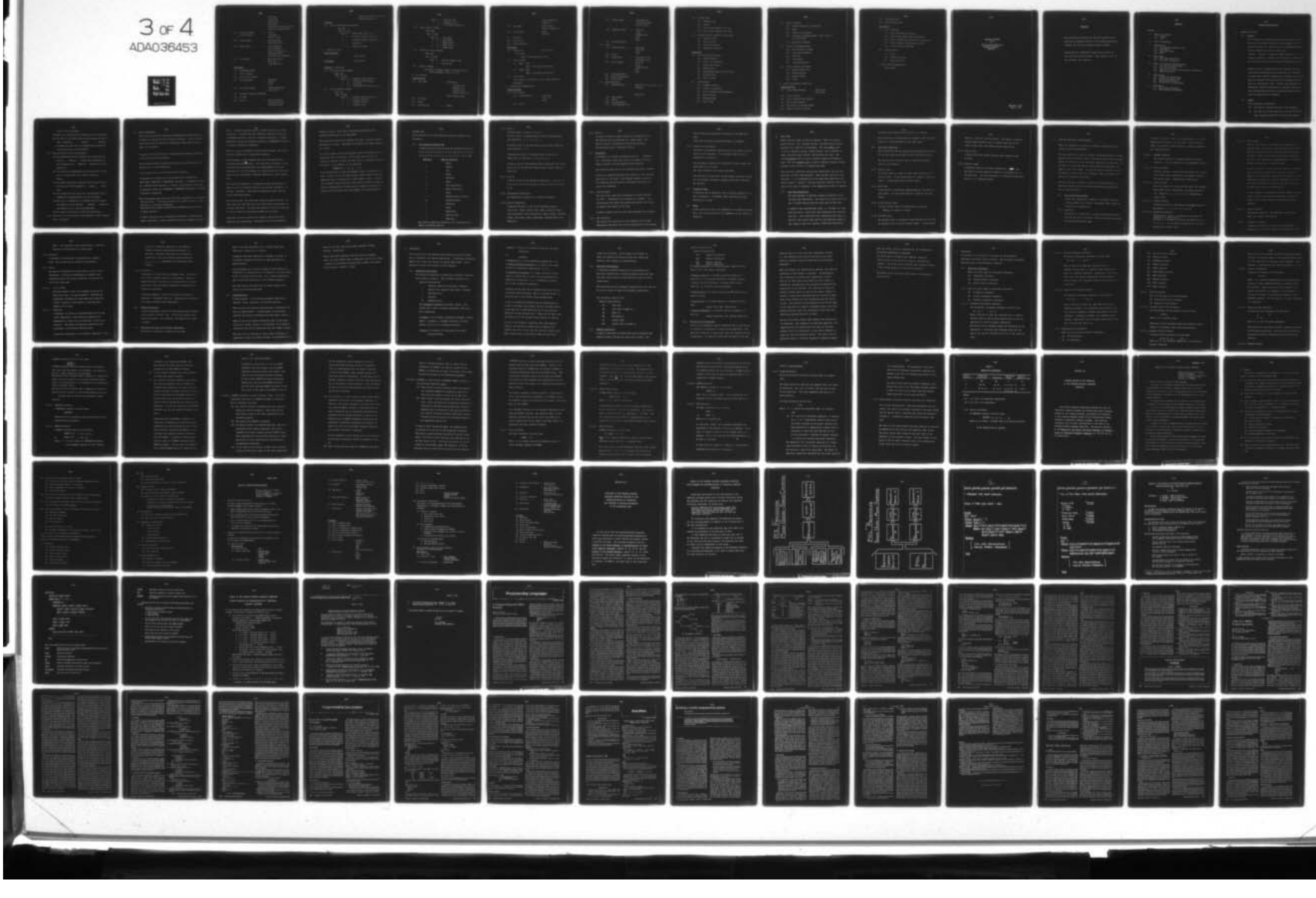
(3) Process Variable Output ID

(4) Control Parameter & Data Source ; Gain
    Reset Time

AD-A036 453 PURDUE UNIV LAFAYETTE IND PURDUE LAB FOR APPLIED IND--ETC F/G 9/2
SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIO--ETC(U)
JAN 77 N00014-76-C-0732

UNCLASSIFIED NL

3 OF 4
ADA036453



Rate Time
Derivative
Filter Gain
Dead Time
Gap Width (Minimum Output)
Other Non-linear Factor

(5) Set-Point Value & Data Source ; Value
Loop Name
Cascade Level

(6) Valve Status ; Direct/Reverse
Feed Back Point

(7) Limit Check ; Loop Condition (On,Off,Fault)
Rate of Change Limit
Feed Back Input
Required Action/Message

(8) Information ; Loop Description
Eng. Unit
Loop Name (No.)

Procedures

(1) Triggering Order

(2) Rate of Update

(3) Base Time of Update

(4) Loop Actuation ; Automatic
Manual
Hold

(5) Set Point Change ; Increase/Decrease
Data Source

(6) Parameter Change & Tracking

(7) Storage

(8) Option ; Error Condition
Feed Back Action
Recovery Action

Replace Loop With Alternate
Manual Manipulation

Commands

(1) Select Standard DDC Algorithm

ALG type
Exp. of type
Style a : Signal add, split, etc.
" b : Standard transfer function
" c : Standard Function Generator
" d : Comparator
" e : Optional element

3. Sequence Control

Specifications

Under Study

Procedures Under Study

Commands (Reference)

(1) Status Detecting Command

SDET type
Exp. of type
Style a : Sequence Step Initialize
" b : Process Status Change
" c : Interrupt
" d : Intermediate Variable Change

(2) Action Request Command

RQST type
Exp. of type
Style a : Sequence Step Initialize
" b : Sequence Execute
" c : Sequence Monitor
" d : Sequence Hold

Style e : Sequence Stop
" f : Sequence Skip
" g : Call Special Function

(3) Timer Active Command

STIM type

Exp. of type

Style a : Set Timer
" b : Read Timer
" c : Timer Active
" d : Timer Hold
" e : Timer Delay

(4) Transfer Control Command

STRN type

Exp. of type

Style a : Branch Sequence Step

" b : Return End

(5) Logical Command

Same Type in Digital Input of Characteristics
(Third Workshop, Part 1, P. 80)

4. File Builder

Specification

(1) File Type ; System File
  Variable File
  Message File
  Constant File
  System Common File

User's Defined File

(2) File Name

(3) Device

(4) Organization ; Random

Index Sequential
Sequential

(5) File Type ; Fixed Length
Dynamic Expandable

(6) Record Type ; Fixed
Variable

(7) File Size

(8) Record Size

(9) File Protection

Procedures

(1) Define File File Set Up

DEFN type

type = Characteristic of File

(2) Direct Access

(a) READ type

type = File Name and Position

(b) WRITE type

type = File Name and Position

(3) File Access

Under study for file open, close, read and write procedure.

5. Man-Machine Communication

Specifications

(1) Data Display

(a) Identification ; Data Type
Point No.
Time

(b) Device

- (c) Display Mode ; Instantaneous
  - Continuous Update
  - Cyclic Point
- (d) Display Format ; Point ID
  - Value
  - Decimal Point
  - Unit
  - Text
- (3) ID Entry Method

(2) Data

- (a) Identification ; Data Type
  - Point No.
  - Time
- (b) Device
- (c) Value Limit ; High Limit Value
  - Low Limit Value
- (d) Entry Format ; Point ID
  - Value
  - Decimal Point
  - Unit
  - Text
- (e) ID Entry Method
- (f) Value Entry Method
- (g) Display Mode

(3) Trend Recording

- (a) Identification ; Point No. 1 (e. g. Pen, bus, display)
  - .
  - .
  - .
  - Point No. n
- (b) Time Marker
- (c) Device
- (d) Recording Mode
- (e) Start Message Code

(4) Message Form

- (a) Message Code
- (b) Device
- (c) Data Point Sequence for logs
- (d) Data Format Sequence for logs

(5) Request Function (operator functions)

- (a) Function Code
- (b) Branch Vector
- (c) Receive Condition
- (d) Terminate Condition

Procedures

(1) Active and Passive Entry

- (a) Request Receive
- (b) Action on Completion
- (c) Entry Data Receive
- (d) Display Data
- (e) Acknowledge input and Store Data
- (f) Message Output
- (g) Cyclic Action

(2) Alarm Display

- (a) Trigger Condition
- (b) Action on Completion
- (c) Error Condition Acknowledgement
- (d) Alarm Display
- (e) Message Output
- (f) Clear Alarm

(3) Graphic Display
  (a) Pattern Generation (Selection)
  (b) Order
  (c) Action on Completion
  (d) Selection of Display Mode; Vector Mode or Character Mode
  (e) Repetitive Display

(4) Control Loop Manipulation
  (a) Loop Condition Change
  (b) Loop Status Display
  (c) Loop Add
  (d) Loop Delete
  (e) Loop Modification

(5) Memory Display and Change
  (a) Memory Address Set
  (b) Memory Display
  (c) Memory Change
  (d) Print

6. User's Maintenance (optional)

Specifications

(1) Error Identification ; Error Code
  Error Type

(2) Error Priority

(3) Error Condition Description

(4) Error Branch Vector

(5) Maintenance Block and Action

(6) Replace Device or Action

(7) Recovery Action

(8) Error Message Code

Procedures

(1) Error Handling

   (a) Error Detection and Branch
   (b) Save Status (Mark Down by Computer)
   (c) Device Failure Set (Mark Down by Operator)
   (d) Device Replace (Alternative Device)
   (e) Device Restart (Mark Up)

(2) Program Debug

   (a) Trace
   (b) Conditional Dump
   (c) Break Point Set

7. Inter System Communication

Under Study

WORKING DOCUMENT
FOR
THE SPECIFICATION OF A PROBLEM ORIENTED LANGUAGE

PWFSOICL, POLC
March, 1971

FOREWORD

This description presents the form for and the interpretation of programs written in the common programming language for use on industrial computer systems.

Suggestions for improvement gained in the review of this material will be welcome. They should be sent to the Chairman, POL Committee.

CONTENTS

SECTION

1. Purpose and Scope
   - 1.1 Purpose
   - 1.2 Scope

2. Basic Terminology

3. Program Form
   - 3.1 The Language Character Set
   - 3.2 Statements
   - 3.3 Statement Label
   - 3.4 Names

4. Data Types
   - 4.1 Data Type Association
   - 4.2 Data Type Properties

5. Data and Procedure Identification
   - 5.1 Data and Procedure Names
   - 5.2 Function Reference
   - 5.3 Type Rules for Data and Procedure Identifiers
   - 5.4 Dummy Arguements

6. Expressions
   - 6.1 Arithmetic Expressions
   - 6.2 Relational Expressions
   - 6.3 Logical Expressions
   - 6.4 Evaluation of Expressions

7. Statements
   - 7.1 Executable Statements
   - 7.2 Nonexecutable Statements

LANGUAGE DESCRIPTION

1. PURPOSE AND SCOPE

1.1 Purpose

This description establishes the form for and the interpretation of programs expressed in the language for the purpose of promoting a high degree of interchangeability of such programs for use on a variety of computer systems. A processor shall conform to this standard provided it accepts, and interprets as specified, at least those forms and relationships described herein.

Insofar as the interpretation of the language form and relationships described are not affected, any statement of requirement could be replaced by a statement expressing that the standard does not provide an interpretation unless the requirement is met. Further, any statement of prohibition could be replaced by a statement expressing that the description does not provide an interpretation when the prohibition is violated.

1.2 Scope

This description establishes:

(1) The form of a program written in the language.

(2) The form of writing input data to be processed by such a program operating on industrial computer

systems (to be defined).

(3) Rules for interpreting the meaning of such a program.

(4) The form of the output data resulting from the use of such a program on computer systems, provided that the rules of interpretation establish an interpretation (to be defined).

1.2.1 This description does not prescribe:

(1) The mechanism by which programs are transformed for use on a computer system (the combination of this mechanism and computer system is called a processor).

(2) The method of transcription of such programs or their input or output data to or from a computing medium.

(3) The manual operations required for set-up and control of the use of such programs on computer equipment.

(4) The results when the rules for interpretation fail to establish an interpretation of such a program.

(5) The size or complexity of a program that will exceed the capacity of any specific computer system or the capability of a particular processor.

(6) The range or precision of numerical quantities.

1.2.2 The Intra- and Inter-Program relationships of this language shall be defined.

2. BASIC TERMINOLOGY

This section introduces some basic terminology and some concepts. A rigorous treatment of these is given elsewhere. Certain conventions concerning the meaning of grammatical forms and particular words are presented.

A program that can be used as a self-contained computing procedure is called an executable program.

An executable program consists of precisely one main program and possibly one or more subprograms.

A main program is a set of statements and comments.

A subprogram is similar to a main program but is headed by a BLOCK DATA, FUNCTION, or SUBROUTINE statement. A subprogram headed by a SPECIFICATION statement is called a specification subprogram. A subprogram headed by a FUNCTION or SUBROUTINE statement is called a procedure subprogram.

[2] The term program unit will refer to either a procedure or a specification program. [1] The term procedure will refer to either a main-program or a procedure subprogram.
Any program unit except a specification subprogram may reference a global procedure.

A global procedure that is defined by statements is called a procedure subprogram. Global procedures also may be defined by other means. A global procedure may be a global function or a global subroutine. A global function defined by statements headed by a FUNCTION statement is called a function subprogram. A global subroutine defined by statements headed by a SUBROUTINE statement is called a subroutine subprogram.

There is a type of statement called a NOTE statement that merely provides information for documentary purposes.

The statements in this language fall into two broad classes - executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement.

The syntactic elements of a statement are names and operators. Names are used to reference objects such as data, procedures, or devices. Operators , including the imperative verbs, specify action upon named objects.

One class of name, the array name, deserves special mention. An array name must have the size of the identified array defined in an array declarator. An array name qualified only by a subscript is used to identify a particular element of the array.

Data names and the arithmetic (or logical) operations may be connected into expressions. Evaluation of such an expression develops a value. This value is derived by performing the specified operations on the named data.

The identifiers used in the language are names. Data are named. Procedures are named. Statements are labeled with names followed by a colon.

At various places in this description, there are statements with associated lists of entries. In all such cases the list is assumed to contain at least one entry unless an explicit exception is stated. As an example, in the statement

SUBROUTINE s $a_1, a_2, \ldots a_n$

it is assumed that at least one symbolic name is included in the list. A list is a set of identifiable elements each of which is separated from it successor by a comma and/or a space. Further, in a sentence a plural form of a noun will be assumed to specify also the singular form of that noun as a special case when the context of the sentence does not prohibit this interpretation.

3. PROGRAM FORM

Every program unit is constructed of characters grouped into statements.

3.1 The Language Character Set

A program unit is written using the following characters:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	Name of Character
	Space
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slant
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point or Period
$	Currency Symbol
;	Semicolon
:	Colon
'	Apostrophe
?	Question Mark
_	Underline

The order in which the characters are listed does not imply a collating sequence.

3.1.1 Digits:

A binary digit is either a 1 or a 0.

An octal digit is a binary digit or one of the following six characters: 2, 3, 4, 5, 6, 7.

A decimal digit is an octal digit or one of the following two characters: 8, 9.

A hexadecimal digit is a decimal digit or one of the additional six characters: A, B, C, D, E, F.

Strings of digits representing numeric quantities are interpreted in the decimal base number system, unless otherwise specified.

3.1.2 Letters:

A letter is one of the twenty-six characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

3.1.3 Alphanumeric Characters:

An alphanumeric character is a letter or a digit.

3.1.4 Special Characters:

A special character is one of the following sixteen characters: Space, Equals, Plus, Minus, Asterisk, Slant, Left parenthesis, Right parenthesis, Comma, Period, Currency Symbol, Semi-colon, Colon, Apostrophe, Question Mark, and Underline.

3.1.5 Spaces:

A string of spaces occupying consecutive columns has the same meaning for the Processor as a single space. Any special character excluding the underline character, surrounded by spaces, represents itself.

3.2 Statements

A statement is a string of characters of the character set. Statements consisting entirely of spaces are called empty statements and are ignored by the Processor.

The character positions in the statement are termed columns.

Columns are numbered consecutively starting at the left and proceeding to the right. The leftmost column is column 1. The rightmost column contains a semi-colon and marks the end of the statement.

3.2.1 NOTE Statement:

The first five significant characters of a note statement are NOTE: designating the statement as a comment. This statement does not affect the program and may be used freely to improve the clarity of the text.

A NOTE statement may not be the last statement of a program.

3.2.2 END Statement:

The significant characters of the statement must be END; designating the end of the written description of the program.

Every program must physically terminate in the END statement.

There must be exactly one END statement in a program.

3.2.3 Ordering of Statements:

Statements are numbered consecutively in the order of presentation to the Processor. The statement which is first presented is statement 1.

Two non-empty statements are equivalent if they contain the same ordered set of words.

Two empty statements are always equivalent.

Two programs are equivalent if they contain the same ordered set of non-empty statements (assuming that note statements are ignored).

3.3 Statement Label

A statement may be labeled so that it may be referred to in other statements. A statement label consists of a name followed by a colon.

3.4 Names

A name consists of one to 32 alphanumeric or underline characters, the first of which must be alphabetic or the underline character.

4. DATA TYPES

Eight different types of data are defined. These are integer, scaled integer, real, extended integer, extended scaled integer, extended real, logical, and character. The term numeric data is used to indicate integer, scaled integer, real, extended integer, extended scaled integer, and extended real data. The term non-numeric data is used to indicate logical and character data. The term extended type is used to indicate extended integer, extended scaled integer or extended real data.

Each type has a different mathematical significance and may have different internal representation. Thus the data type has a significance in the interpretation of the associated operations with which a datum is involved. The data type of a function defines the type of the datum it supplies to the expression in which it appears.

4.1 Data Type Association

The name employed to identify a datum or function carries the data type association. The form of the string representing a constant defines both the value and the data type.

A symbolic name representing a function, variable, or array must have only a single data type association for each program unit. Once associated with a particular data type, a specific name implies that type for any differing usage of that symbolic name that requires a data type association throughout the program unit in which it is defined.

Data type must be established for a symbolic name by declaration in a type-statement for all data types.

4.2 Data Type Properties

The mathematical and the representation properties for each of the data types are defined in the following sections.

For numeric data the value zero is considered neither positive nor negative.

4.2.1 Integer Type:

An integer datum is always an exact representation of an integer value. It may assume positive, negative, and zero values. It may only assume integral values.

4.2.2 Real Type:

A real datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values.

4.2.3 Scaled Integer Type:

A scaled integer datum is second form of real data.

(Radixpt is implicit in data)

4.2.4 Extended Type:

An extended datum is a processor approximation to the value of integer, real, or scaled integer number. It may assume positive, negative, and zero values. The degree of approximation though undefined must be greater than that of the types integer real, and scaled integer respectively.

4.2.5 Logical Type:

A logical datum may assume only the truth values of true or false.

4.2.6 Character Type:

A character datum is a string of characters. T[illegible]ring may consist of any characters capable of representation, in the processor. The blank character is a valid and significant character in a character datum.

5. DATA AND PROCEDURE IDENTIFICATION

Names are employed to reference or otherwise identify objects, including data and procedures.

The term reference is used to indicate an identification of a datum implying that the current value of the datum will be made available during the execution of the statement containing the reference. If the datum is identified but not necessarily made available, the datum is said to be named. One case of special interest in which the datum is named is that of assigning a value to a datum, thus defining or redefining the datum.

The term reference is used to indicate an identification of a procedure implying that the actions specified by the procedure will be made available.

5.1 Data and Procedure Names

A data name identifies a constant, a variable, an array or array element, or a block. A procedure name identifies a function or a subroutine.

5.1.1 Constants:

A constant is a datum that is always defined during execution and may not be redefined. Rules for writing constants are given for each data type.

A numeric constant is said to be signed when it is written immediately following a plus or minus. Also, for these types, an optionally signed constant is either a constant or a signed constant.

5.1.1.1 Integer Constant

An integer constant is a binary constant, an octal constant, a decimal constant, or a hexadecimal constant.

5.1.1.1.1 Binary Constant

A binary constant is written as the letter "B" followed by a single quote, followed by a non-empty string of the digits 0 and 1, followed by a single quote.

5.1.1.1.2 Octal Constant

An octal constant is written as the letter "O" followed by a single quote, followed by a non-empty string of the digits 0, 1, 2, 3, 4, 5, 6, 7, followed by a single quote.

5.1.1.1.3 Decimal Constant

A decimal constant is written as a non-empty string of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

5.1.1.1.4 Hexadecimal Constant

A hexadecimal constant is written as the letter "X" followed by a single quote, followed by a non-empty string of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and the letters A, B, C, D, E, F, followed by a single quote.

5.1.1.2 Real Constant

A basic real constant is written as an integer part, a decimal point, and a decimal fraction part in that order. Both the integer part and the decimal part are strings of digits; either one of these strings may be empty but not both. The constant is an approximation to the digit string interpreted as a decimal numeral.

A decimal exponent is written as the letter, E, followed by an optionally signed integer constant. A decimal exponent is a multiplier (applied to the constant written immediately preceding it) that is an approximation to the exponential form ten raised to the power indicated by the integer written following the E.

A real constant is indicated by writing a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent.

5.1.1.3 Logical Constant

The logical constants, true and false, are written TRUE and FALSE respectively.

5.1.1.4 Character Constant

A character constant is written as a single quote followed by a string of characters followed by a single quote. Two sequential single quotes within a character constant is interpreted as a single quote.

5.1.2 Variable:

A variable is a datum that is identified by a symbolic name. Such a datum may be referenced and defined.

5.1.3 Array:

An array is an ordered set of data of one, two, or three dimensions. An array is identified by a symbolic name. Identification of the entire ordered set is achieved via use of the array name.

5.1.3.1 Array Element

An array element is one of the members of the set of data of an array. An array element is identified by immediately following the array name with a qualifier, called a subscript, which points to the particular element of the array.

5.1.3.2 Subscript

A subscript is written as a parenthesized list of subscript expressions. Each subscript expression is separated by a comma from its successor, if there is a successor. The number of subscript expressions must correspond to the declared dimensionality.

A subscript expression is an arithmetic expression. The value of a subscript expression is the greatest integer contained in the evaluated arithmetic expression. Following evaluation of all subscript expressions, the array element successor function (____________) determines the identified array element.

5.1.4 Procedures:

A procedure is identified by a symbolic name. A procedure is a local or global function or subroutine. Local and global functions are referred to as functions or function procedures; local and global subroutines as subroutines or subroutine procedures.

A function supplies a result to be used at the point of reference; a subroutine does not. Functions are referenced in a manner different from subroutines.

5.2 Function Reference

A function reference consists of the function name followed by an optional argument list. If the list contains more than one argument , the arguments are separated by commas or spaces.

5.3 Type Rules For Data and Procedure Identifiers

The type of a constant is implicit in its name.

There is no type associated with a symbolic name that identifies a subroutine or a block.

A symbolic name that identifies a variable, an array, or a statement function must have its type specified in a type-statement.

In the program unit in which a global or local function is referenced, its type definition is defined in the same manner as for a variable and an array. For a function subprogram, type is specified explicitly in the FUNCTION statement.

The same type is associated with an array element as is associated with the array name.

5.4 Dummy Arguments

A dummy argument of an external procedure identifies a variable, array, subroutine, or external function.

When the use of an external function name is specified, the use of a dummy argument is permissible if an external function name will be associated with that dummy argument .

When the use of an external subroutine name is specified, the use of a dummy argument is permissible if an external subroutine name will be associated with that dummy argument .

When the use of a variable or array element reference is specified, the use of a dummy argument is permissible if a value of the same type will be made available through argument association.

Unless specified otherwise, when the use of a variable, array, or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

6. EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic, relational, and logical expressions. A relational expression appears only within the context of logical expressions. An expression is formed from elements of the language.

6.1 Arithmetic Expressions

An arithmetic expression is formed with arithmetic operators and arithmetic elements. The arithmetic operators are:

Operator	Representing
+	Addition, positive value (zero + element)
-	Subtraction, negative value (zero - element)
*	Multiplication
/	Division
**	Exponentiation

The arithmetic elements are primary, factor, term , signed term, simple arithmetic expression, and arithmetic expression.

A primary is an arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

A factor is a primary or a construct of the form:

primary**primary

A term is a factor or a construct of one of the forms:

term/factor

or:

term*term

A signed term is a term immediately preceded by + or -.

A simple arithmetic expression is a term or two simple arithmetic expressions separated by a + or -.

An arithmetic expression is a simple arithmetic expression or a signed term or either of the preceding forms immediately followed by a + or - immediately followed by a simple arithmetic expression.

A primary of any type may be exponentiated by an integer or extended integer primary, and the resultant factor is the same type as the element being exponentiated.

A real or extended real primary may be exponentiated by a real or extended real primary, and the resultant factor is of type real if both primaries are of type real and otherwise of type extended real. These are the only cases for which use of the exponentiation operator is defined.

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element and the resultant element will have a range equal to the maximum range of any of its constituent elements. By the range of an element is meant the absolute difference between the largest and smallest value capable of representation.

6.2 Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator and will have the value true or false as the relation is true or false, respectively.

The evaluation of the relational expression will use the larger of the ranges of these two arithmetic expressions.

The relational operators are:

Operator	Representing
LT	Less than
LE	Less than or equal to
EQ	Equal to
NE	Not equal to
GT	Greater than
GE	Greater than or equal to

6.3 Logical Expressions

A logical expression is formed with logical operators and logical elements and has the value true or false. The logical operators are:

Operator	Representing
OR	Logical disjunction
AND	Logical conjunction
NOT	Logical negation
EOR	Logical comparison

The logical elements are logical primary, logical factor, logical term, and logical expression.

A logical primary is a logical expression enclosed in parentheses, a relational expression, a logical constant, a logical variable reference, a logical array element reference, or a logical function reference.

A logical factor is a logical primary or NOT followed by a logical primary.

A logical term is a logical factor or a construct of the form:

logical term AND logical term

A logical expression is a logical term or a construct of the form:

logical expression OR logical expression

6.4 Evaluation of Expressions

A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence (except as modified in the following paragraph).

When two elements are combined by an operator, the order of evaluation of the elements is optional. If mathematical use of operators is associative, commutative, or both, full use of these facts may be made to revise orders of combination, provided only that integrity of parenthesized expressions is not violated. The value of an integer or extended integer factor or term is the nearest integer or extended integer whose magnitude does not exceed the magnitude of the mathematical value represented by that factor or term. The associative and commutative laws do not apply in the evaluation of integer or extended integer terms containing division, hence the evaluation of such terms must effectively proceed from left to right.

Any use of an array element name requires the evaluation of its subscript. The evaluation of functions appearing in an expression may not validly alter the value of any other element within the expressions, or any other statement in which the function reference appears. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguements or subscript.

No factor may be evaluated that requires a negative valued primary to be raised to a real or extended real exponent. No factor may be evaluated that requires raising a zero valued primary to a zero valued exponent.

No element may be evaluated whose value is not mathematically defined.

7. STATEMENTS

A statement is classified as executable or non-executable. Executable statements specify actions. Non-executable statements describe objects and attributes of objects.

7.1 Executable Statements

There are three types of executable statements:

(1) Assignment statements
(2) Program Control statements
(3) Device Control statements

7.1.1 There are three types of assignment statements:

(1) Arithmetic statements
(2) Logical assignment statements
(3) Character assignment statements

7.1.1.1 Arithmetic Assignment Statement

The arithemetic assignment statement is of the form:

$$V_1 = V_2 = . . . V_n = e$$

Where n must be at least one, and each V_1 is a numeric variable name or numeric array element name of the same type and e is in an arithmetic expression.

Execution of this statement causes the evaluation of the expression e, converting the resulting value into the type of V_1, and the altering of all V's to the converted value.

7.1.1.2 Logical Assignment Statement

The logical assignment statement is of the form:

$$V_1 = V_2 = . . . \ V_n = e$$

Where n must be at least one, and each V_1 is a logical variable name or a logical array element name, and e is a logical expression. Execution of this statement causes the logical expression e to be evaluated and its value to be assigned to each of the logical entities V_1's.

7.1.1.3 Character Assignment Statement

The character assignment statement is of the form:

$$V_1 = V_2 = ... = V_n = e$$

Where n must be at least one, and each V_1 is either a character variable , or a character array element, and e is either a character constant consisting of a single character, a character variable, or a character array element. Execution of this statement causes each of the V_1's to assume the value of e.

7.1.2 Program Control Statements:

There are twelve program control statements:

(1) GO TO Statements

(2) IF statements

(3) Subroutine call statement

(4) RETURN statement

(5) CONTINUE statement

(6) REPEAT statement

(7) PAUSE statement

(8) PAUSE UNTIL statement

(9) WHEN statement

(10) CANCEL statement

(11) STOP statement

(12) START statement

7.1.2.1 GO TO Statements

There are two types of GO TO statements:

(1) Unconditional GO TO statement

(2) Computed GO TO statement

7.1.2.1.1 Unconditional GO TO Statements.

An unconditional GO TO statement is of the form:

GO TO k

Where k is a statement label.

Execution of this statement causes the statement identified by the statement label to be executed next.

7.1.2.1.2 Computed GO TO Statement.

A computed GO TO statement is of the form:

$$\text{GO TO } (k_1, k_2, \ldots k_n), i$$

where the k's are statement labels and i is an integer variable reference.

Execution of this statement causes the statement identified by the statement label k, to be executed next, where j is the value of i at the time of the execution. This statement is defined only for values such that 1 = j = n.

7.1.2.2 Logical IF Statement

A logical IF statement is of the form:

IF (e) S

Where e is a logical expression and S is any executable statement except a REPEAT statement or another logical IF statement. Upon execution of this statement, the logical expression e is evaluated. If the value of e is false, statement S is executed as though it were a CONTINUE statement. If the value of e is true, statement S is executed.

7.1.2.3 Subroutine Call Statement

A subroutine call statement is of the form:

S; or

S ℓ;

Where ℓ is a list of actual arguements.

The inception of execution of a call statement references the designated subroutine. Return of control from the designated subroutine completes execution of the call statement.

7.1.2.4 RETURN Statement

A RETURN statement is one of the forms:

RETURN ;
RETURN d;

A RETURN statement marks the logical end of a procedure and, thus, may only appear in a procedure. The argument d must be a dummy argument of the procedure.

Execution of this statement when it appears in a subroutine causes return of control to the current calling procedure and, if d was specified, it is assumed the d is associated with a statement label.

Execution of this statement when it appears in a function causes return of control to the current calling procedure.

At this time the value of the function is made available.

7.1.2.5 CONTINUE Statement

A CONTINUE statement is of the form:

CONTINUE ;

Execution of this statement causes continuation of the normal execution sequence.

7.1.2.6 REPEAT Statement

A REPEAT statement has one of the forms:

REPEAT TO ℓ $i = m_1, m_2, m_3$;

or REPEAT TO ℓ $i = m_1, m_2$;

Where: 1) ℓ is the label of a CONTINUE statement.

This CONTINUE statement is the terminal

statement of the associated REPEAT, and must physically follow and be in the same program unit as that REPEAT statement.

2) i is an integer variable name; this variable is called the control variable.

3) m_1, called the initial parameter; m_2, called the terminal parameter; and m_3, called the incrementation parameter, are each either an integer constant or integer variable reference. If the second form of the REPEAT statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the REPEAT statement, m_1, m_2, and m_3 must be greater than zero.

Associated with each REPEAT statement is a range that is defined to be those executable statements from and including the first executable statement following the REPEAT, to and including the terminal statement associated with the REPEAT. A special situation occurs when the range of a REPEAT contains another REPEAT statement. In this case, the range of the contained REPEAT must be a subset of the range of the containing REPEAT.

A completely nested nest is a set of REPEAT statements and their ranges, and any REPEAT statements contained within their ranges, such that the first occurring terminal statement of any of those REPEAT statements physically follows the last occurring REPEAT statement and the first occurring REPEAT statement of the set is not in the range of any REPEAT statement.

7.1.2.6.1 A REPEAT statement is used to define a loop. The action succeeding execution of a REPEAT statement is described by the following six steps:

(1) The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

(2) The range of the REPEAT is executed.

(3) If control reaches the terminal statement, then after execution of the terminal statement, the control variable of the most recently executed REPEAT statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

(4) If the value of the control variable after incrementation is less than or equal to the value represented

by the associated terminal parameter, then the action described starting at step 2 is repeated, with the understanding that the range in question is that of the REPEAT, whose control variable has been most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter, then the REPEAT is said to have been satisfied and the control variable becomes undefined.

(5) At this point, if there were one or more other REPEAT statements referring to the terminal statement in question, the control variable of the next most recently executed REPEAT statement is incremented by the value represented by its associated incrementation parameter and the action described in step 4 is repeated until all REPEAT statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

In the remainder of this section (7.1.2.6) a logical IF statement containing a GO TO is regarded as a GO TO.

(6) Upon exiting from the range of a REPEAT by the execu-

tion of a GO TO statement; that is, other than by satisfying the REPEAT, the control variable of the REPEAT is defined and is equal to the most recent value attained as defined in the preceding paragraphs.

7.1.2.6.2 A REPEAT is said to have an EXTENDED RANGE if both of the following conditions apply:

(1) There exists a GO TO statement within the range of the innermost REPEAT of a completely nested nest that can cause control to pass out of that nest.

(2) There exists a GO TO statement not within the nest that, in the collection of all possible sequences of execution in the particular program unit, could be executed after a statement of the type described in (1), and the execution of which could cause control to return into the range of the innermost REPEAT of the completely nested nest.

If both of these conditions apply, the extended range is defined to be the set of all executable statements that may be executed between all pairs of control statements, the first of which satisfies the condition of (1) and the second of (2). The first of the pair is not included in the extended range; the second is. A GO TO statement may not cause control to pass into the range of a REPEAT unless it is being executed as part of the extended range of that particular REPEAT. Further, the extended range of a REPEAT may not contain a REPEAT of the same program unit that has an extended range. When a procedure reference occurs in the range of a REPEAT, the actions of that procedure are considered to be temporarily within that range; i.e., during the execution of that reference.

The control variable, initial parameter, terminal parameter, and incrementation parameter of a REPEAT may not be redefined during the execution of the range or extended range of that REPEAT.

If a CONTINUE statement is the terminal statement of more than one REPEAT statement, the statement label of that terminal statement may not be used in any GO TO that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

7.1.2.7 Pause Statement

The Pause statement is of the form:

```
PAUSE  n u ;
```

Where n is an integer variable or constant, and u is either SECONDS, MINUTES, OR HOURS.

The PAUSE statement causes normal program flow to be suspended at the point of execution. The program flow is restarted after the indicated time interval n u has elapsed. When this time interval has elapsed, the program flow is restarted at the statement following the PAUSE statement.

7.1.2.8 PAUSE UNTIL Command

A PAUSE UNTIL command is of the form:

```
PAUSE UNTIL  ( e ) ;
```

Where e is a logical expression

The PAUSE UNTIL command causes the normal program flow to be suspended at the point of execution. The program flow is restarted when the expression e is TRUE. When the expression becomes TRUE, the program flow is restarted at the statement following the PAUSE UNTIL command.

7.1.2.9 WHEN Statement

A WHEN statement is of the form:

ℓ : WHEN (e) S ;

Where e is a logical expression, and S is an executable statement, except WHEN; ℓ is a statement label.

The execution of the WHEN statement does not affect normal program flow. However, if following the execution of statement ℓ the expression e becomes true, the normal program flow is interrupted and statement S is executed. Following the execution of S normal program execution is resumed without loss of continuity. A WHEN statement remains active from the point of execution until it is negated by a CANCEL statement.

7.1.2.10 CANCEL Statement

The CANCEL Statement is of the form:

CANCEL ℓ ;

Where ℓ is a statement label. The execution of this statement cancels the WHEN statement whose label is e.

7.1.2.11 STOP Statement

The STOP statement is of the form:

STOP ;

or STOP p ;

Where p is a program name.

If used alone (STOP), this statement terminates the execution of the program in which it is located. Therefore, it should be located at the logical end of the program. One or more than one such STOP statements can be used in a program.

If used with the arguement p (STOP p), this statement terminates the execution of program p.

called a type statement.

7.2.2 Array-Declarator:

An array declarator specifies an array used in a program unit.

The array declarator indicates the symbolic name, the number of dimensions, (one, two, or three), and the size of each of the dimensions. The array declarator form may be in a type-statement.

An array declarator has the form:

v (i)

Where: (1) v, called the declarator name, is a symbolic name.

(2) (i), called the declarator subscript, is composed of 1, 2, or 3 expressions, each of which may be an integer constant or an integer variable name. Each expression is separated by a comma from its successor if there are more than one of them. In the case where i contains no integer variable, i is called the constant declarator subscript.

The appearance of a declarator subscript in a declarator statement serves to inform the processor that the declarator name is an array name. The number of subscript expressions specified for the array indicates its dimensionality. The magnitude of the values given for the subscript expressions indicates the maximum value that the subscript may attain in any array element name.

No array element name may contain a subscript that, during execution of the executable program, assumes a value less than one or larger than the maximum length specified in the array declarator.

7.2.3· Array Element Successor Function and Value of a Subscript:

For a given dimensionality, subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain are indicated in Table 2. A subscript expression must be greater than zero.

The value of the array element successor function is obtained by adding one to the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and has no successor element.

TABLE 2

VALUE OF A SUBSCRIPT

Dimensionality	Subscript Declarator	Subscript	Subscript Value	Maximum Subscript Val.
1	(A)	(a)	a	A
2	(A, B)	(a, b)	a + A (b - 1)	A B
3	(A, B, C)	(a, b, c)	a + A (b - 1) + A B (c - 1)	A B C

NOTES:

(1) a, b, and c are subscript expressions.

(2) A, B, and C are dimensions.

7.2.4 Integer Statement:

An INTEGER statement is of the form:

$$\text{INTEGER } v_1, v_2, v_3 \ldots v_n$$

Where v_1 is either a variable name or an array declarator.

TO BE DEFINED FOR ALL OBJECTS.

SECTION VII

JAPANESE REVIEW OF THE PROPOSALS OF THE PROBLEM-ORIENTED LANGUAGES COMMITTEE

The several Japanese committees studying the field of industrial computer systems and software have been extremely helpful to the several workshops even before the formation of the Japanese Regional Branch of the International Purdue Workshop on Industrial Computer Systems. This material represents some of their contributions to the work of the Problem-Oriented Language Committee. The material appeared in the *Minutes of the Fourth and Fifth Workshops on Standardization of Industrial Computer Languages*, pp. 203-207 and 24-27 respectively.

PRECEDING PAGE BLANK-NOT FILMED

November 1970

COMMENTS TO THE PROBLEM ORIENTED LANGUAGE COMMITTEE

Working Group for the Problem
Oriented Languages
Technical Committee on Industrial
Computer Languages of Japan
Chairman, T. TOHYAMA

1. Present Status of Problem Oriented Language

At present, a few computer manufacturers have been developed the program packages for industrial computer applications, but these packages are utilized for specific application area and do not designed so much systematic consideration for the generalized process control oriented language.

We recognize the importance and the necessity of standardization of problem oriented language and hope further expansion of this work.

It is user's basic requirement to design computer control system as possible as easy and economic, and to improve system at plant-site. Besides, the standardization of specification for application is important to get effectiveness in system development and system maintenance of various computers.

In Japan, the trend of standardization of problem oriented language is not enhanced sufficiently. But, we find the necessities and the activities of standardization of industrial computer language through workshops at Purdue University.

We have Working Group for the Problem Oriented Language and study your reports of workshop. Our group just started to investigate the requirement and the specification of problem oriented language. Herein, we show the comments and our idea to the report of the Problem Oriented Language Committee at the 3rd Workshop.

PRECEDING PAGE BLANK-NOT FILMED

2. Comments

Followings are comments and requirements to develop the standardization of Problem Oriented Language (POL).

(1) It is the 1st step of work to describe the detail specifications of hardware and software interfaces for industrial computer system. In 2nd step, procedures and syntax are established and next step Programs or Languages themself are defined.

(2) Form of POL has function of self documentation and easy understanding.

(3) The conceptual computer system is basis for the establishment of POL. If this is basic requirement, it is important to describe the complete characteristics of computer system.

(4) Following requirements are options of POL developer for user's convenience and attractiveness.
   (a) Building block structure for expansion of applications
   (b) Quick response
   (c) Small requirement of memory
   (d) Parallel I/O operation
   (e) On-line program development and maintenance
   (f) Multiprogramming in multi-task

(5) It is necessary to provide Level and/or Subset of POL with each functions or structure in languages.

(6) POL is supervised under Operating System (OS). But, POL itself can handle interrupts of hardware and software, and error handling.

(7) POL will be more rigorously defined below.
   (a) Relation of structure between Procedural Languages and POL
   (b) Calling sequence between each languages
   (c) Mixed programming by using each languages
   (d) File access method

(8) It is better to define the level of computer size for generating the object programs from the source documentations.

(9) It is convenience for future study to define the basic configuration for running the object programs.

(10) Before translation of POL, the following procedure are declared.
   (a) Hardware Assignment
   (b) Data Type Definision
   (c) Core Allocation
   (d) File Allocation

3. Scope of User's Function in POL.

The following requirements of POL are put in order. We have idea to establish POL to co-operate your workshop.

There are seven sub-functions proposed under our working group.

(Detail informations are shown in Attachment.)

(1) Variables
   (a) Process Variable Input
   (b) Process Variable Output
   (c) Process Status Input
   (d) Process Status Output
   (e) User's Interrupt
   (f) Intermediate Variable

(2) DDC

  (a) Loop definition

  (b) Standard DDC Algorithm

  (c) Easy construction of required control algorithm

(3) Sequence Control

  (a) Standard Sequence Control Algorithm

  (b) Easy construction of required logic flow chart

At present, we do not define the followings.

- Description of Logic Flow Chart for computer
- Description of request, abnormal condition for control
- Specification for Input/Output
  (Using Process Status Input and Output in Variables)

(4) File Builder

  (a) Same file structure in POL's file and User's file

  (b) Easy file access from user's file

(5) Man-Machine Communication

  (a) Communication mode

  - Active Action
  - Passive Action
  - Alarm Display

  (b) Message Format definition

  (c) Expandability for new device

(6) User's Maintenance

  (a) Error definition (Process side)

  (b) Error Handling

(7) Inter Communication

  (a) Computer to Computer

  (b) Remote Peripheral Device to Computer

  (c) POL and Procedural Language

April 1971

COMMENTS TO PROBLEM ORIENTED LANGUAGE

Working Group for the Problem
Oriented Languages
Technical Committee on Industrial
Computer Languages of Japan
Chairman, T. TOHYAMA

1. Outline our goals for the year

For 1971, we have established the same objectives for Problem Oriented Languages as our primary concern's:

a. Develop Specification and Procedure for Sequence Control, Inter Systems Communication and Process Control Function.

b. Review and exploit The Preliminary Recommended Specification and Procedure for the Process Oriented Languages.

c. Develop the shorthand form (Fill in the Blanks System) on Process Oriented Languages.

d. Investigate the necessity and the requirement for application programs in the industrial computer.

2. Supplements for Specification and Procedure for a Process Oriented Language

Followings are supplementary specification and procedure for the fourth workshop on standardization.

(A) Sequence Control

Specifications

(1) Sequence Block ; Name

(2) Sequence Step Type ; Digital Logic
Decision Table
Analog Logic
Timmer
Trigger

(3) Variable Input ID ; Logical Input
Process Variable Input
Interrupt

(4) Variable Output ID ; Logical Output
Process Variable Output

(5) Parameter ; Intermediate Variable
Decision Rule
Control Limit

(6) Data Source ; Global
Local
Historical
Multiple

(7) Faile Safe Condition ; Normal Condition
Hold Condition
Emergency Condition
Answer Back Point

(8) Alarm Condition ; Abnormal Condition
Required Action/Message

(9) Information ; Sequence Block Description
Sequence Block Name
Sequence Step Name List
Step Operation Descriptions
Step Operation Condition
Block-out Operation No.

Procedures

(1) Order of Sequence Block

(2) Order of Sequence Step

(3) Action on Completion of Sequence Step

(4) Action on Completion of Sequence Block

(5) Interactive Operation of Sequence

(6) Rate of Update in Repetitive Operation

(7) Detect Status

(8) Action Request ; Initialize
Execute
Monitor
Hold
Skip
Stop
Call Special Function

(9) Timer Action ; Set
Read
Active
Hold
Delay

(10) Transfer Control

(11) Logical and Arithmetic Operation

(12) Parameter Change & Write Status

(13) Store

(14) Option ; Emergency Shutdown
Standby Shutdown
Restart
Partial Auto/Partial Manual

(B) Inter Systems Communication

At present, the following approach for the Inter Systems Communication is performed to prepare the specification and procedure.

(1) Computer to Computer
  a. Data Communication Mode
  b. Communication Line control
  c. Message Text
  d. Error Check

(2) Inter Language
  a. Longhand Form and Shorthand Form
  b. Problem Oriented Languages and FORTRAN
  c. Problem Oriented Languages and Long Term Procedural Languages
  d. Problem Oriented Languages and Assembler

(3) Function of Hierarchy System
  a. Hierarchy System
  b. Dual System/Duplex System

(C) Process Variable Input from On-line Analyzer
(Reference for Gaschromatograph)

Specification

(1) Signal Type

(2) Procedure to Input ; Device Address
Terminal Address
Processing Sequence

(3) Stream Identification ; Stream Code

(4) Component Identification ; Component Code
Value Address

(5) Read Status ; Instrument Status
Read Ready Condition
Gain Selection

(6) Validation ; Validity Check
Required Action/Message

(7) Conversion Parameter

(8) Filtering ; Filter Type
Filtering Factor

(9) Limit Check ; Instrument Limit
Process Variable Limit
Rate of Change Limit

(10) Information ; Variable Description
Eng. Unit
Variable Name

Procedure

(1) Read Mode

(2) Select Stream

(3) Detect Instrument Status

(4) Select Gain

(5) Action on Completion

(6) Rate of Update for Calculation

(7) Adjust Base point

(8) Calculate Component's Value

(9) Average and Min. Max.

(10) Storage

(11) Option ; Change Instrument
Read Stop
Recovery Action
Replace Action Variable

SECTION VIII

DECISIONS OF THE PROBLEM-ORIENTED LANGUAGES COMMITTEE RELATING TO THE TRANSLATOR METHOD OF PROCEEDING AND PERTINENT LITERATURE REFERENCES TO THE TECHNIQUES USED

At the end of 1971 the Problem-Oriented Languages Committee decided upon the Problem-Oriented Language Converter and Translator as their desired end product (Avenue 2 described earlier). The attached documents from the *Minutes of the Sixth Workshop on Standardization of Industrial Computer Languages*, Chapter IV, pp. 65-74, and the *Minutes of the Seventh Workshop*, Chapter IV, pp. 91, 96-97, and 102-134. The latter pages reproduce several of the definitive papers by Professor W. M. Waite of the University of Colorado, on STAGE 2, the basic work in the translator area.

REPORT OF THE PROBLEM ORIENTED LANGUAGES COMMITTEE SIXTH WORKSHOP ON STANDARDIZATION OF INDUSTRIAL COMPUTER LANGUAGES

Based upon the results of the 10th Meeting of the Committee attached herein and of further discussion during this Workshop the POL Committee has adopted the following resolution concerning its future work:

> The Pol Committee will now concern itself with defining the facility (POLCAT STRUCTURE) which will allow the generation of POL's that are compatible with the LTPL and that all have the same basic structure.

To illustrate this proposal the Committee has worked out the defining example of Figures 1-4 as illustrations of the work contemplated.

It is estimated by the Committee that this effort will require the equivalent of one man year of work.

If four Committee meetings are held each year (two at the Workshops and two at intermediate times) for an average of two days with five participants, the necessary man years of effort can be accomplished in five years.

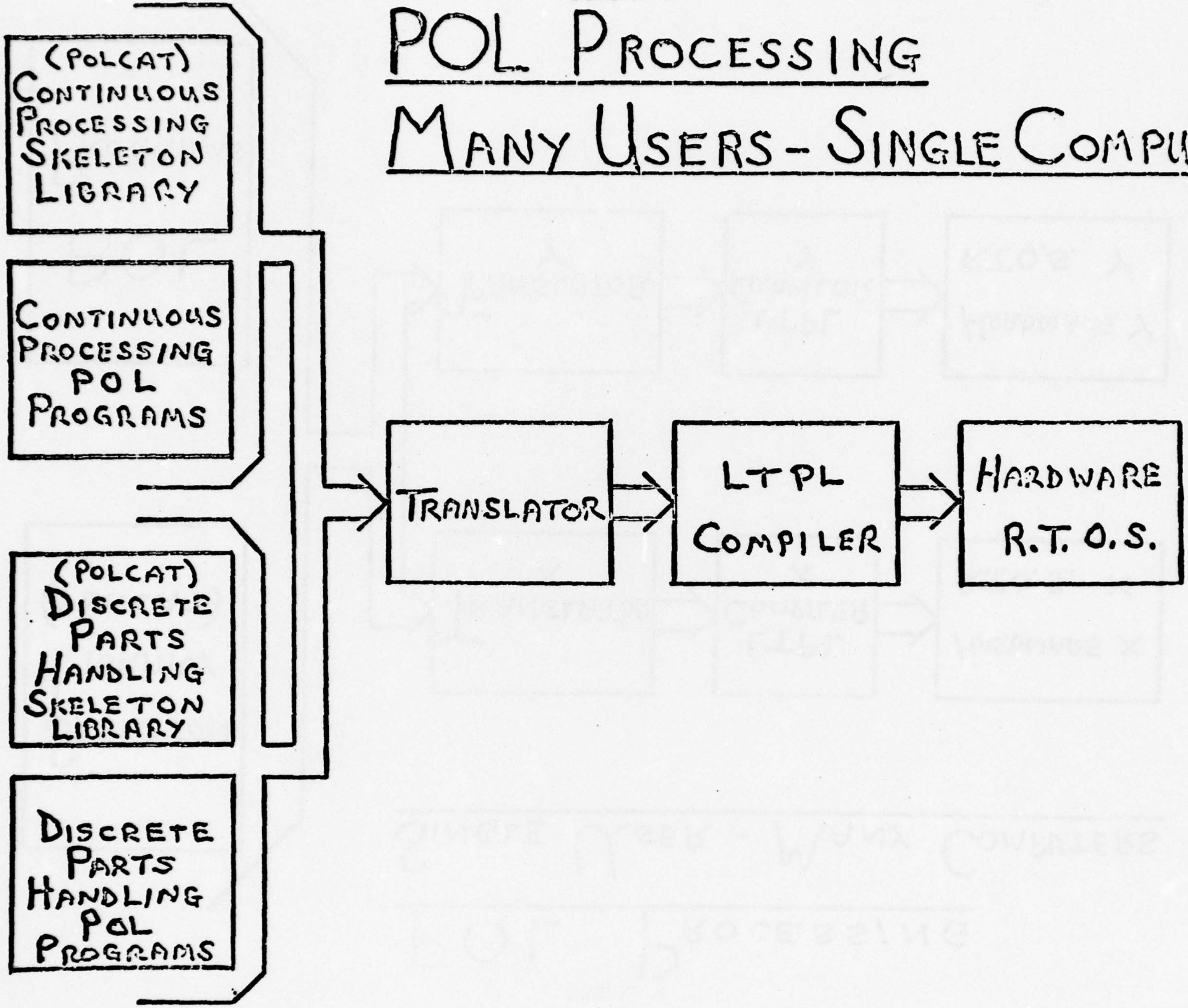
Therefore the Committee appeals to the Workshop Attendees for increased participation in its work to reduce this overall time period of its project.

PRECEDING PAGE BLANK-NOT FILMED

FIGURE 1

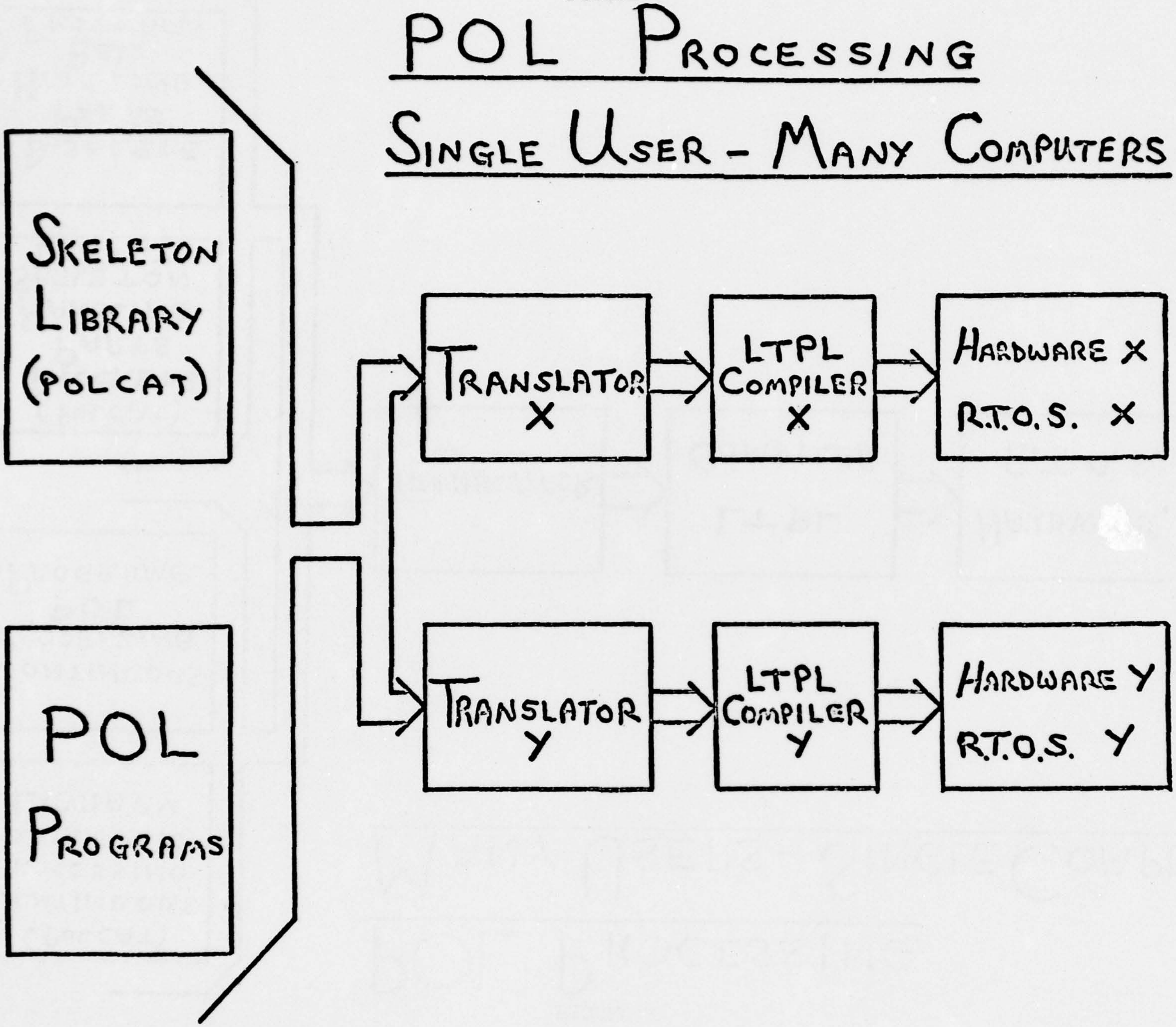
POL PROCESSING

MANY USERS - SINGLE COMPUTER


(POLCAT) CONTINUOUS PROCESSING SKELETON LIBRARY
CONTINUOUS PROCESSING POL PROGRAMS
(POLCAT) DISCRETE PARTS HANDLING SKELETON LIBRARY
DISCRETE PARTS HANDLING POL PROGRAMS
TRANSLATOR
LTPL COMPILER
HARDWARE R.T. O.S.

PRECEDING PAGE BLANK-NOT FILMED

FIGURE 2



POL PROCESSING

SINGLE USER - MANY COMPUTERS

SKELETON LIBRARY (POLCAT)
POL PROGRAMS
TRANSLATOR X
LTPL COMPILER X
HARDWARE X
R.T.O.S. X
TRANSLATOR Y
LTPL COMPILER Y
HARDWARE Y
R.T.O.S. Y

FIGURE 3

PROBLEM ORIENTED LANGUAGE CONVERTER AND TRANSLATOR

- STATEMENT TYPE POLCAT EXPANSION

```
SCAN X1 THRU X100 EVERY 1 SEC;
```

```
PØLCAT
KEY 'SCAN'
FØRMAT DELIM(1,' ')
FØRMAT TERM (;)
FØRMAT ARG (1,TYPE1), ARG (2,TYPE1), ARG(3,TYPE2), ARG(4, TYPE3)
       ØRDER  KEY, DELIM(1), ARG(1), DELIM(1),'THRU', DELIM(1)
              ARG(2), DELIM(1),'EVERY', DELIM(1), ARG(3)
              DELIM(1), ARG(4), TERM
BECØMES

     [ LTPL DATA SPECIFICATIONS       ]
     [ AND/OR PROGRAM STATEMENTS      ]

END
```

FIGURE 4

PROBLEM ORIENTED LANGUAGE CONVERTOR AND TRANSLATOR

- FILL-IN-THE-FORM TYPE POLCAT EXPANSION

BLOCK TYPE 1 = SCAN 2 = CONTROL 3 = LOG	10	1
STARTING POINT	20	X1
FINAL POINT	25	X100
FREQUENCY	30	1
UNITS 1 = SEC 2 = MIN 3 = HOUR	40	1

```
PØLCAT
KEY `1'
FØRMAT  FIELD(1,10), FIELD(2,20:24), FIELD(3,25:29) FIELD(4,30:34)
        FIELD (5,40)
FØRMAT  ARG(1,TYPE1), ARG(2,TYPE2), ARG(3,TYPE2), ARG(4,TYPE1)
        ØRDER BY FIELD  KEY, ARG(1), ARG(2), ARG(3), ARG(4)
BECØMES

        [ LTPL DATA SPECIFICATIONS
          AND/OR PROGRAM STATEMENTS ]

END
```

MINUTES - 10th MEETING OF THE PROBLEM ORIENTED LANGUAGES COMMITTEE

Foxboro, Massachusetts Sept 23-24, 1971

by

N. P. Wilburn

Attendees:
P. Wilburn - WADCO Corporation
D. Hartmann - Bailey Meter Company
E. Bristol - Foxboro Company
R. Flath - Leed's and Northrup

Opening Review

To begin the meeting a review was made of the minutes of the previous POL Meeting[1] and the "Summary of the POL Committee Activities".[2] The study of this material revealed that, just exactly what a POL should include, was still unresolved.

Characteristics of a POL

The committee then tried to answer the question "what is the definition of a POL and what are its characteristics", by looking at three sources:

1. The POL Committee Minutes, Summary, etc.
2. Vendor Statements and Positions
3. The Attendee's Understanding

The following observations were made in this discussion:

- The POL should enhance transfers of data between sections of the computer (as defined on Page VI-2 of Reference 2).
- The POL should be like a set of tools that the design engineer can use.
- The computer should be transparent.
- The POL is equated with a user oriented language where the user is not a programmer.
- User should be able to set down in a POL, a solution to his problem.
- Transportability of programs that was needed in the business and scientific programs is not required in the POL.
- But universality of POL's is necessary so that retraining for each user instance is not required.

---

[1] "Minutes - 9th Meeting of the POL Committee", Layfayette, Indiana, May 3-6, 1971

[2] "Summary of POL Committee Activities" edited by P. Wilburn, July 1971.

In light of the above discussion the POL Committee came to the following conclusions and concensus:

- There cannot be just one POL (Concluded also in several other POL Committee Meetings)
- The POL should allow both "Fill-in-the-blanks" type programs and "sentence" type.
- It seems an impossible (or at least a very lengthy) task to meet both of the above requirements in a committee situation.
- Therefore the POL Committee should confine itself to describing the basic architectural structure of the POL language(s).
- Certain generally applicable built in functions should be made available in the POL and the specifications for these will be made by the committee.
- The POL will have the LTPL as its basis and the POL will accept into its structure all of the LTPL definitions such as character sets, etc.
- The POL Committee acknowledges the possibility of logical inconsistencies resulting between the POL and the LTPL.
- The POL will include on line real time commands to allow the user to debug with certified algorithms or modules.

The most important conclusion made was:

> ... The POL Committee will henceforth concern itself with defining the structure of a "macroing" capability to be added to the LTPL (when defined). With this structure it will be possible to generate many POL's all having the same basic structure whose "sentences" or "fill-in-the-blanks" tables can be expanded into the LTPL (similar in manner to macro expansions done in assembly languages).

Macro Structure

To give the discussion a basis, the example POL statement suggested in the sixth meeting of the POL Committee was again selected:

UPDATE n EVERY t, a FOLLOWING id STARTING b TH SECOND:

A rough structure for a possible means of defining the macro was made: (underscored words are assumed to be reserved).

```
MACRO BEGIN
    DECLARE KEY STRING 'UPDATE'
    FORMAT DELIM (1,' ') (2,',')
        TERMINATOR (;)
        ORDER KEY, DELIM(1), ARG(1), STRING 'EVERY',
            DELIM(1), ARG(2), DELIM(2), STRING 'FOLLOWING'
            ARG(3), DELIM(1), STRING 'STARTING',
            . . . . . . . . . . . . . . . . . . . . . . . . . .
        ARG(1, 25 CHAR, NAME)
        ARG(2, 3 DIGIT, TIME)
        ARG(3, 5 CHAR, NAME)
    BECOMES
        CALL System Prog #3 (ARG1, ARG2, ARG3)
        . . . . . . . . . . . . . . . . . . . . . . . . . . .
    END
```

Where the reserved words serve the following roles:

MACRO	Identifies macro and possibly changes meanings of some reserved words from LTPL to POL.
BEGIN	Beginning of Macro
DECLARE	Declarations beginning
KEY	Identifies STRING which triggers macro
FORMAT	Begins structural description of macro calling sequence
DELIM	Specifies delimiters to be used in macro
TERMINATOR	Specifies terminating character of macro
ORDER	Specifies order of macro call

STRING	Specifies strings to be used in macro call
ARG	Specifies arguments to be used in macro call
BECOMES	Indicates beginning of LTPL program into which macro is to be expanded.

In discussing this macro structure the following conclusions were reached:

- Deviations from the specified order of the macro result in differing possibilities
    1. Substitution of default values
    2. Error message
    3. Alternate code
- The POL will only allow deletions from the macro order not rearrangements (eg. deletion of "EVERY t," from example)
- POL will have reserved words (eg. ORDER, DELIM)
- There will be only one type of macro structure
- POL syntax will be imbedded in LTPL syntax
- Macro names will not be used as variables
- Communication between macro expansions at compile time will be made using computer tables
- Conditionals will be allowed in the macro expansion

REPORT OF THE PROBLEM ORIENTED LANGUAGES COMMITTEE

SEVENTH WORKSHOP ON STANDARDIZATION OF INDUSTRIAL COMPUTER LANGUAGES

The POL Committee met informally on Tuesday morning during the Seventh Workshop. The committee discussed the following items:

A. The procedures as to how one can obtain STAGE2 should be included in the minutes. -- STAGE2 or the Mobile Programming System Package can be obtained upon request from:

Graduate School Computing Center
Univeristy of Colorado
Boulder, Colorado 80302

The package consists of:

1. W. M. Waite - Technical Report 71-10. ($3.00)
2. W. M. Waite - Technical Report 69-2. ($2.60)
3. W. M. Waite - Technical Report 69-3. ($4.75)
4. W. M. Waite - Technical Report 69-3B ($3.40)
5. W. M. Waite - "Implementation Guide for Mobile Programming System." ($2.00)
6. 026 Card Deck for Mobile Programming System. ($32.00)
7. (Alternate) Industry Compatible 7-Track Magnetic Tape for Mobile Programming System. ($18.00)

Approximately one month is required between request and receipt of package.

B. That the POL Committee meet informally at each Workshop to review current status and whether the POL Committee should be reactivated.

C. The possibility of having another presentation to the Workshop (maybe on a fifth day) by vendors on the characteristics of their particular POL or procedural language.

D. The possibility of presentations of user experiences with implementation of STAGE2.

E. Copies of References 2, 3, 5, 6, 7, and 8 of the Bibliography of these Minutes are included here.

Operated by the
Westinghouse
Hanford Company
for the USAEC

A Subsidiary of
Westinghouse Electric
Corporation

Hanford Engineering Development Laboratory

P.O. Box 1970
Richland, Washington
99352

March 6, 1972

PROBLEM ORIENTED LANGUAGES COMMITTEE MEETING

The next meeting of the POL committee of the Purdue Workshop on Standardization of Industrial Computer Languages will be Thursday and Friday, March 23 and 24 in LaJolla, California at the Holiday Inn beginning Thursday at 9:00 AM.

Fred Ruebhausen has agreed to handle room reservations for anybody who needs them. He can be reached at the following address or phone.

F. E. Ruebhausen
Control Data Corporation
4455 Eastgate Mall
LaJolla, California 92037
(714) 453-2500 ext. 325

As you may have observed on Pages 65 to 75 of the last workshop minutes (attached), the direction of the POL committee's effort has changed somewhat. We will be concerned for the most part at LaJolla with defining "POLCAT." I believe the following references are pertinent to this effort and it would be helpful if you could look at them (if available) before the meeting.

1. "Texas Instrument Language Translator," Texas Instruments, Incorporated, Manual #955382-9701 (December 1971)
2. "A Language Independent Macro Processor," William M. Waite, Communications of the ACM, Vol. 10, No. 7, July 1967
3. "A Base for a Mobile Programming System," Richard J. Orgass and William M. Waite, Communications of the ACM, Vol. 12, No. 9, September 1969
4. "Macro Instruction Extensions of Compiler Languages," M. Douglas McIlroy, Communications of the ACM Vol. 3, April 1960
5. "Syntax Macros and Extended Translation," B. M. Leavenworth, Communications of the ACM, Vol. 9, No. 11, November 1966
6. "Building a Mobile Programming System," W. M. Waite, The Computer Journal, Vol. 13, No. 1, February 1970
7. "The ML/I Macro Processor," P. J. Brown, Communications of the ACM, Vol. 10, No. 10, October 1967

March 6, 1972

8. "The Mobile Programming System: STAGE2," W. M. Waite, Communications of the ACM, Vol. 13, No. 7, July 1970

I am looking forward to seeing as many of you as possible in LaJolla.

Pat

N. P. Wilburn
Chairman, POL Committee

NPW/se

Programming Languages

N. WIRTH, Editor

A Language Independent Macro Processor

WILLIAM M. WAITE
University of Colorado, Boulder, Colorado*

A macro processor is described which can be used with almost any source language. It provides all features normally associated with a macro facility, plus the ability to make arbitrary transformations of the argument strings. The program is used at the Basser Computing Department, University of Sydney, Sydney, Australia, to process text for eight different compilers.

1. Introduction

The term "macro" was first used to denote a feature of certain assembly languages which allowed a programmer to refer to a group of instructions as though they were a single instruction. By mentioning the name of the "macro-instruction," the programmer caused all of the component instructions to be inserted at that point in his coding. The possibilities of this sort of redefinition were soon realized, and a classic paper by McIlroy [1] showed how an appropriate set of macro instructions could allow the user to tailor an assembly language to his needs at a quite high level. Until recently little more was said on the subject, and most of the operations proposed by McIlroy were used routinely in a number of assembly languages. The problem of reprogramming has been responsible for a resurgence of interest, and several papers [2-5] have appeared during the past two years.

One of the significant features of the current papers is the concept of a macro processor which is independent of any particular assembly language. Macro processing is viewed as a type of string manipulation—a character stream fed to the input is analyzed according to certain rules, and a character stream is produced as output. The processor described in this paper is designed to operate as such a string manipulator, and its output is to be presented to some compiler or assembler. Because it can deal with almost any input text, it has been named LIMP (*L*anguage-*I*ndependent *M*acro *P*rocessor).

In classical macro processors such as that described by McIlroy, a macro definition has the form:

MACRO NAME (P_1, P_2, $\cdots$, P_n)
Code body
END

Here the words "MACRO" and "END" serve to delimit the definition, "NAME" is the macro name, and "P_1" through "P_n" are formal parameters. The code body is a symbol string which may contain instances of the formal parameters. The form of a call on this macro would be "NAME (A_1, A_2, $\cdots$, A_k)" where $k \leq n$. Such a call is replaced by the code body of the macro "NAME", with the actual parameter strings A_1, A_2, $\cdots$, A_k being substituted for the corresponding formal parameters P_1, P_2, $\cdots$, P_k. If $k < n$, then the processor generates actual parameter strings for P_{k+1}, $P_{k+2}, \cdots, P_n$ in some regular way.

LIMP generalizes the notion of a macro definition and macro call. The line which introduces a macro definition may be any arbitrary character string, with parameters indicated by a special parameter symbol. We refer to this line as a *template*. A template essentially allows the name part of a traditional macro to be replaced by a "distributed name" consisting of all portions of the line other than the parameter markers. This approach frees the system from any arbitrary format restrictions of a particular language—templates can be written in a form suited to the language at hand.

Macro calling is accomplished by pattern matching against the set of templates defined by the user. Any line which cannot be matched is copied directly to the output without change. When a match occurs between an input line and a template, the code body corresponding to the template is evaluated with the actual parameters resulting from the template match. The result of this evaluation replaces the matched line in the output text.

Correspondences between actual and formal parameters are set up during template matching. The template is a sequence of fixed strings separated by "holes" (parameter markers). When the matching process is complete, each parameter marker will correspond to some substring of

---

This work was performed at the Basser Computing Department, University of Sydney, Sydney, Australia, and was supported by the U.S. National Science Foundation under Postdoctoral Research Fellowship No. 45070.

* Department of Electrical Engineering, University of Colorado

the input line; the fixed strings of the template will exactly match other substrings of the line. The string matched by a given parameter marker becomes the actual parameter corresponding to that given formal parameter.

The code body of a LIMP macro consists of a series of statements in a language almost identical to SNOBOL [6, 7]. Formal parameters are specified by their relative addresses in the template, up to 9 formal parameters being allowed per macro. (The relative address is a digit between 1 and 9 inclusive, which specifies the number of the parameter marker in the template, counting from the left.) Whereas conventional macro processors permit only one form of substitution of actual for formal parameters, LIMP allows several alternative forms. The type of substitution is specified for a given instance of an actual parameter by a subscript on the relative address.

The structure of LIMP allows it to be used as a preprocessor for almost any compiler or assembler. It was incorporated into the KDF9 operating system developed by the Basser Computing Department in April, 1966 [8] and since then has been used with programs written in eight languages (three variants of ALGOL, two assembly codes, a list processor, a flowcharting language, and an autocode). No change in LIMP is required for any of these languages.

The input to LIMP consists of a series of lines. (On card-oriented machines each card is a line; on paper tape, lines are delimited by carriage-return characters). The first line defines the set of control characters for the run. This *flag line* is necessary to preserve the language independence of the processor; different languages generally require different control characters.

The first character of the flag line is the *end-of-line flag*. If it appears on any subsequent line, its effect is that of a carriage return—the remainder of the line is ignored by LIMP. This allows the use of any text as commentary on any line. If "space" is specified as the end-of-line flag, LIMP assumes that *no* end-of-line character is desired.

The second character of the flag line is the parameter marker. It is only recognized as a control character when a template is being entered into the set of macro definitions; at any other time it is treated as a normal character. In this paper, the parameter flag will be represented by "*", and the end-of-line character by ".".

After the flag line, the user defines his macros. At least one definition must follow the flag line, and further definitions may appear anywhere in the text. Each macro must, of course, be defined before it is called.

The free format of the input line and the large number of templates involved create possible ambiguities in the matching process. Section 2 describes the template-matching algorithm in detail and shows how the scanner resolves these ambiguities. An informal specification of the code body statements can be found in Section 3, where the various types of parameter conversions are defined. This section gives a brief review of the features of SNOBOL which are crucial to the understanding of LIMP. In Section 4 a number of examples of LIMP macros are given, chosen to illustrate the basic principle and some of the unique features of the language. Section 5 outlines the construction of the processor and gives an indication of the direction of future work on this project.

2. Template Matching

Matching a single template against an input line is a special case of the following general problem [9]: "Given a pattern $\{c_1c_2 \cdots c_n\}$ and a string S, locate consecutive substrings s_i in S such that each s_i is an acceptable value of the pattern element c_i." Pattern elements in LIMP templates are either fixed strings or parameter flags. Each element can be assigned a *weight*, w_i, equal to the length of the shortest acceptable value of c_i. If c_i is a fixed string, w_i is its length; if it is a parameter flag, $w_i = 0$ (because a parameter flag can match the null string).

In general we shall have a whole set of patterns which are candidates for the matching process. These patterns may be grouped into a tree with each element corresponding to a branch; the branches leaving a given node can then be ordered according to the weights of the corresponding elements.

Often there may be several ways for an input line to match a given template, and a given input line might match any one of several templates. For example, the line "X = Y = Z." would match any of the templates "* = *.", "X = *.", "X = Y = *." and many more. Moreover, it could match "* = *." in two ways: With actual parameter 1 equal to "X = Y", or with actual parameter 1 equal to "X". Such possibilities complicate the matching process and require further specifications to eliminate ambiguity.

In order to define uniquely which template will match a given line, and which of the possible matches to this template will be chosen, the following rules are used.

Rule 1. A space which is not adjacent to a parameter flag matches any nonempty string of spaces, and spaces at the end of a line are ignored unless the last character of the matching template is a parameter flag.

Rule 2. The matching process proceeds from left to right with each pattern element matching the shortest possible substring.

Rule 3. At a node, matches are attempted for the branches in order of decreasing element weights.

Rule 4. If no element at a node can match a substring, then a new match is attempted at the previous node. This new match is accomplished by extending the substring formerly matched to the next shortest acceptable value for the same element. If this extension cannot be made, a match for the next element at that node is attempted. If no element remains, rule 4 is applied to that node.

The pattern match succeeds when the last character of the input line has been matched; it fails when no match can be found for the first character.

As an example of the matching process, consider the set of templates

(1) SAM = A.
(2) SAM = *.
(3) * = *.
(4) * = A.
(5) * = * = *.

The tree for this set is shown in Figure 1, where a lower case "b" is used to denote a space.

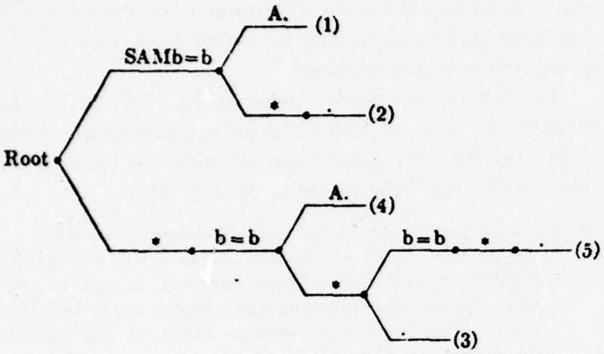


Fig. 1. An example of a template tree

The input line "SAM = A." will match template (1), as will any line which has a nonempty string of spaces between "SAM" and "=" or between "=" and "A.". The effect of Rule 1 is thus to make a string of spaces significant, but the exact number of spaces in the string is immaterial. (When a template is read and merged with the tree any string of spaces is replaced by a single space.)

Suppose that the next two input lines were "SAM = B." and "SAM =   C.", where there are six spaces between "=" and "C" in the second line. Both lines match template (2), and in the first line the value of the actual parameter will be "B". In the second line, however, the value of the actual parameter will be "   C"—five of the six spaces have been included in the actual parameter. This is because the space in template (2) between "=" and "*" is adjacent to a parameter flag: A space adjacent to a parameter flag matches exactly one space, so that the remaining spaces must be absorbed by the parameter. No spaces are actually deleted from the input line during the matching process, so that if no match is found all spaces are preserved when the string is written out.

At first glance one might assume that the input line "B = C = A." would match template (4) with "B = C" as the value of the actual parameter. Consideration of the rules will show, however, that this line will be matched to template (5).

The crucial decision occurs at the second node of the lower trunk of the tree At this point we have matched "B" to the first parameter flag and "b = b" to the fixed portion of the template, as required by Rule 2 ("B" is the shortest possible matching substring for the first parameter). According to Rule 3 the next match to be attempted must be for "A.".

TABLE I. String Names Available in Limp

Name	*Function*
dc ($1 \leq d \leq 9$, $1 \leq c \leq 3$)	actual parameter *d*, conversion *c*
0i ($1 \leq i \leq 9$)	current value of location counter, minus *i*
IN	effective input line
PR	printing output stream
PU	punching output stream
PT	private tree
ST	symbol tree
TT	template tree

The match for "A." fails because the next character of the line is "C". So we attempt a match for the parameter flag. Now the only way in which we can extend the match for the first formal parameter is by repeated use of Rule 4. But this requires that we be unable to find *any* match at all nodes further along the branch. Since the line matches template (5), we will not get the chance to apply Rule 4 at all.

3. Code Body

When Limp matches an input line to a template, it creates actual parameter strings and then executes a series of statements making up the code body of the macro. The statements of the code body are written in a language which is almost identical in form to Snobol. There are two reasons for this choice: (1) The operations available in Snobol allow very general manipulations of the actual parameters. (2) The syntax of Snobol effectively separates the metacharacters from the strings being manipulated and thus preserves the language independence of the processor.

The general form of a statement in the code body of a Limp macro is:

⟨label⟩⟨string⟩⟨pattern⟩ = ⟨replacement⟩/⟨go-to⟩

As in Snobol, a statement label must begin in the first character position of the line and may consist of any string of letters or digits. The label terminates at the first space; if the first character of the line is a space, it is assumed that the statement has no label. The next constituent of the statement is a string reference. In Snobol a string may be given almost any symbolic name, but in Limp the programmer may use only a fixed set of names as summarized in Table I. The only strings which may be used as working storage during the execution of a code body are those corresponding to the nine possible actual parameters. System strings (such as IN) have special properties which disqualify them for use as working storage. The private tree (PT) can be used to store intermediate results when more than nine strings are needed.

TABLE II. Pattern Elements Accepted by **Limp**

Type	*Format*	*Matches*
Literal	'any string'	itself
Constant	dc $(0 \leq d \leq 9,$ $0 \leq c \leq 9)$	a substring containing the same characters as the specified constant string.
Arbitrary	dA	any string or substring, including the null string
Balanced	dB	a non-void string which is balanced with respect to ( )
Fixed-length	dFk	any string of length k (k may be any number of digits)
Back-referenced	dR	the string which was previously matched to variable d

Essentially this tree constitutes the "backup store" for the macro processor; it is used only by code bodies and is never altered by any part of the system.

The third constituent of the statement, the pattern, may be absent. If present, it consists of a series of pattern elements separated by blanks. The allowed pattern elements are listed in Table II. When a literal string is read, the procedure for handling quotes is as follows: The first quote marks the beginning of the literal. Any subsequent string of n consecutive quotes is replaced by a string of k consecutive quotes, where k is the greatest integer not greater than $n/2$. If there is a nonzero remainder from this division, the literal terminates. A literal may extend over several lines, in which case a carriage return character is inserted in the literal at the end of each line. This allows the programmer to write a single literal which produces several lines of output.

If a pattern is specified in a statement, the pattern is tested against the string named by the string reference. If the pattern can be matched to some substring of the string reference, strings are created for each variable in the pattern. Also, the matching substring may be altered by specifying a replacement. This replacement may contain the variables defined by the pattern match; see [6, 7] for further details. Note that if no replacement is desired, both the "=" and the replacement string should be omitted. If "=" is present but no replacement string is specified, the null string is assumed.

A statement may stretch over several lines. If the line changes occur within a literal, they result in carriage return characters as noted above. If they fall between the constituents of the statement, they are treated as spaces. Constituents must be separated by at least one space or carriage return, and additional spaces or carriage returns are ignored. The "/" preceding the go-to field *must* be present whether the go-to is used or not—it serves notice that the current line is the last one for the current statement.

The go-to field may be absent, but if present it takes one of the following forms:

(⟨label⟩) —transfer control to ⟨label⟩ unconditionally
S(⟨label⟩) —transfer control to ⟨label⟩ if the pattern match was successful
F(⟨label⟩) —transfer control to ⟨label⟩ if the pattern match was not successful.

Both conditional jumps may be present, in either order. Spaces are ignored everywhere in the go-to field.

A code body may contain any number of statements, the last of which is distinguished by the label "END". This last statement need not contain any other constituents, including the slash. Of course it may also be a normal code body statement, in which both the slash and string reference are required.

The conversion digits, denoted by "c" in Table I, describe the way in which the actual parameter string is to be used in each particular substitution instance. The meanings of these digits are $(1 \leq d \leq 9)$:

d0 —use an exact copy of the actual parameter string.
d1 —use an exact copy of the actual parameter string; if this conversion appears in a string reference, any pattern specified must match a substring which begins at the first character of the referenced string. (This is equivalent to "anchored mode" in Snobol.)
d2 —look up the string on the symbol tree and use the value found; if the string is not in the symbol tree, use a null string.
d3 —same as d2, except that if the string is not found, it is added to the symbol tree and given the current value of parameter 0 (the "location counter"). Parameter 0 is incremented by one following this definition.

The symbol tree is established either by the use of type 3 parameter calls or by direct action on the part of the programmer. The symbol tree has the name ST, which can be used as the string reference in any statement. Some caution must be exercised, however, because ST is a tree rather than a string. The pattern must begin with a set of constant elements. This constant portion, consisting of all elements up to the first variable, will be tested against the tree and must match some entry all the way from the root to a leaf. The remainder of the pattern will be tested in the usual way against the string attached to the leaf. Any replacement will only be made to that part of the pattern which is *beyond* the leaf. This constraint allows the programmer to change the definition of a symbol, but not to delete a symbol from the tree. The entire tree may be deleted by "ST = /", and a symbol may be added by "ST = ST ⟨specification of symbol⟩/". If the added symbol is to be defined, a second statement must be used.

The system maintains a location counter which the programmer can use in several ways: (1) Type 3 conversion can be used to assign the current value of the location counter to a symbol. The location counter is automatically incremented by 1 following the assignment. (2) A set of local labels can be obtained by using parameter 0 explicitly. A reference to string 0i is taken as the current value of the location counter minus i. If the programmer uses such constructions, he is responsible for incrementing the location counter by the maximum value of *i*, plus 1, *before* the first such reference is encountered. This can be done by the statement "00 = (00 + 'i' + '1')/".

As shown in the statement above, integer arithmetic can be performed on strings in LIMP. Each expression is enclosed in parentheses and has as its value a string containing the numeric value of the expression. The allowed operators are +, −, * and /, with their usual meanings. Note that there is no possibility of confusion between "/" used as an end-of-statement mark and "/" used as a division operator because the latter appears only within parentheses. Parentheses may be nested to arbitrary depth, and arithmetic operations may continue over as many lines as necessary. Numbers are stored as strings of digits; negative numbers are signed, positive are not.

4. Examples of LIMP Macros

The simplest possible macro operation is one in which a single line is expanded into several lines with parameter substitution. For example, suppose we wish to write an arithmetic statement in an assembly language program on a single address machine (the notation for this assembly language is that of [1]):

```
* = * + *.
END PU = 'FETCH, ' 20 '
ADD ' 30 '
STORE, ' 10 /
```

This template would match a line such as "SAM = JOE + BILL.", and the code body would punch:

```
FETCH, JOE
ADD, BILL
STORE, SAM
```

The system string "PU" is the punch unit, and whenever it is assigned a value, that value is punched (followed by a carriage return). "PR" is the printer, and behaves in the same way. The intermediate carriage returns were supplied by including them in literals.

The next step in complexity is to allow a macro call to be used in the code body of another macro. For example, we might define a complex addition by:

```
Z* = Z* + Z*.
  IN = 'R' 10 ' = R' 20 ' + R' 30 /
END IN = 'I' 10 ' = I' 20 ' + I' 30 /
```

If the line which was matched was "ZSAM = ZJOE + SBILL.", then the first statement would assign the value "RSAM = RJOE + RBILL." to the system string IN. (Note that the end-of-line symbol is automatically supplied.) The effect of this assignment is to save the current state of all strings and call the processor recursively to expand the line whose value is assigned to IN. After the expansion is completed, control returns to the next statement. If IN is used in a pattern or replacement, rather than being assigned a value, its value is the next effective input line (an effective input line may have been generated by a macro at a higher level, as "RSAM ···" in the present example).

In most conventional macro processors, an expanded line is automatically submitted for a re-scan; in LIMP the re-scan is under the control of the programmer. For most applications this causes no difficulty, but there are certain macros in which the programmer may not know whether a line contains a macro call or not. In such instances he would resubmit the questionable line to LIMP by assigning it to IN. If it matched some template, it would be expanded; otherwise it would merely be copied into the punch output stream.

All of the normal conditional assembly operations of most macro generators are available in LIMP by means of suitable statements. We can easily test the value of any argument, make arbitrary changes in any argument, skip statements, loop, etc. The only feature of conventional macro processors which is not obviously available is that of nested definition. This is provided by explicit reference to the template tree by means of the symbol "TT". Statements using TT obey exactly the same rules as those using ST, the symbol tree. Suppose, for example, that we were writing an algebraic language based on single-address assembly code. Suppose further that we demanded that the programmer declare all variables. We could define a macro to inform him of undeclared variables:

```
FETCH, *.
END PR = 'VARIABLE ' 10 ' IS UNDECLARED.'/
```

The addition macro defined above would be altered to expand the FETCH instruction (using IN), rather than just punching it, and this would cause the "Undeclared" message to be printed:

```
* = * + *.
  IN = 'FETCH,  ' 20 /
END PU = 'ADD, ' 30 '
STORE, ' 10 /
```

Now, whenever a variable is declared, we must define a FETCH macro for that variable. The template matching process ensures that the macros containing variable names will take precedence over the one with a parameter flag.

```
VARIABLE *.
  PU = 10 ':  RESERVE, 1' /
  TT = TT 'FETCH, ' 10 /
END TT 'FETCH, ' 10 = 'PU = ''FETCH, ' 10 ''' /
END' /
```

The first statement punches an assembly code operation to reserve one location for the variable and define the symbol. The second statement inserts the new template on TT, while the third provides the definition. Note how the quotes in the definition are supplied—since they are inside a literal, their number must be doubled. The slash on the third line is a part of the literal, not the termination of the rule. The FETCH macro for the variable A would thus have the form:

```
FETCH, A.
  PU = 'FETCH, A' /
END
```

Notice here that the use of FETCH as both a macro name and a target language operation is possible because of the programmer's freedom to submit a line for re-scan or punch it out immediately.

The main reason for requiring that the programmer declare his variables in the preceding example was to ensure that space was reserved. This can be done using the LIMP symbol tree and type 3 conversion without requiring declarations. Instead of defining the fetch macro to print an "Undeclared" message, we write:

```
FETCH, *.
END PU = 'FETCH, V+' 13 /
```

By the definition of type 3 conversion, this macro will punch out "FETCH, V + n" where "n" is the value found in the symbol tree for the variable. If the variable was not in the symbol tree, it would be added and given the current value of the location counter. The location counter would then be incremented by one.

At the end of the program, we would reserve a block of space to hold all variables. A little thought will show that the size of this block is just the value of the location counter at the end of the program. We can therefore define a macro END by:

```
END.
END PU = 'V:  RESERVE. ' 00 '
END' /
```

This macro will replace the programmer's END statement with:

```
V:  RESERVE, k
END
```

This reserves a block of memory of the current length and gives it the name V. The assembler's address arithmetic feature then references each variable within this block.

Suppose, however, that the assembler being used cannot do address arithmetic. LIMP macros can still be written to avoid the need for variable declaration:

```
FETCH, *.
END PU = 'FETCH, V' 13 /
END.
  10 = 00 '.'/
  20 = '0' /
CHK 11 20 '.' /S(END)
  PU = 'V' 20 ':  RESERVE, 1' /
  20 = (20 + '1') /(CHK)
END PU = 'END' /
```

Here each variable is named "Vi" for successive integers *i*. The "END" macro then creates a seris of space reservations, one for each variable. It is interesting to note that there is no penalty attached to specifying a go-to. Because of the way the statements are stored, one with no go-to field is effectively supplied with two go-to's, both to the succeeding statement.

The use of TT allows us to add code to existing macro definitions or change them temporarily. The statement

```
TT ⟨constant pattern⟩ dA = ⟨string⟩ d0 /
```

will add code to an existing macro. We can change a macro temporarily by something like:

```
TT 'TEMPLATE *' dA = ⟨string⟩ /
PT = PT 'SAVEMAC' /
PT 'SAVEMAC' = d0 /
```

The current code body of macro "TEMPLATE *" is saved on the private tree as the value of the symbol "SAVEMAC", and is replaced on the template tree by ⟨string⟩. The code body can be recovered from the private tree by:

```
PT 'SAVEMAC' dA = /
TT 'TEMPLATE *' = d0 /
```

In the current version of LIMP the code body of a macro is not held as a string because of the consequent slowdown in interpreting it. It is converted into a list structure when first defined, and hence it is not possible to use string operations to make changes *within* the code body. If such operations are necessary, the code body must be entered by means of "IN" and stored on the private tree as a string. Changes can then be made, and the code body redefined from this string. The following macro will read a code body and store it in the private tree as a string:

```
ENTER    CODE BODY * UNTIL *.
         PT = PT 10    /
         30 =          /
LINE     40 = IN /
         41 20         /S(END)
         30 = 30 40    /(LINE)
END PT   10 = 30       /
```

A call on this macro might be:

```
ENTER CODE BODY SAM UNTIL SAM IS FINISHED.
⟨code body⟩
SAM IS FINISHED.
```

The code body would be read and stored in actual parameter 3 until the line "SAM IS FINISHED" was recognized by the test "41 20 /S(END)". At this point the string from parameter 3 would be stored as the value of "SAM" on the private tree. This string could then be manipulated in any way and defined as the code body of any template.

5. Implementation and Extension

The entire LIMP processor is written in the list language WISP [10, 11], and hence possesses a certain amount of machine as well as language independence (WISP systems are available on the IBM 7094, 7040, GE 265, English Electric KDF9, Elliot 803, EDSAC 2 and Atlas 2). The processor itself is imbedded in an environment which interacts with the operating system to route LIMP output text to the correct compiler. In the Basser system, the compiler to be used is specified by 3 characters of a 12-character program identifier. The user may provide a program title as well as an identifier, and the LIMP environment uses the first 3 characters of this title to replace the compiler specification. The output text from Limp is then resubmitted to the system by writing it on a magnetic tape and resetting the input unit. There is no restriction on the compiler selected; it could easily be Limp again. It is therefore possible to make multiple passes through Limp before going to a conventional compiler.

The version of Limp described in this paper is a reformulation of that which is currently in use in the Basser Computing Department. The two versions are equivalent in power, but the Basser version requires that the code bodies be written out in a form much closer to that in which they are executed. The resulting macro definitions are relatively clumsy to write, and difficult to decipher when debugging.

The processor has two main phases: definition and expansion. When a macro is defined, its template is added to the template tree and its code body is converted into a form which can be handled by an interpretive routine during the expansion phase. This conversion recognizes a number of special cases, and sets up different structures for each. For example, if "PU" is the string reference, then the replacement is formed into a single string with markers for the parameter substitutions. Thus the string need not be re-evaluated each time the macro is expanded. Successive punch statements are combined, with suitable insertion of carriage return characters. Similar simplifications can be made where the string reference is "PR" or "IN". Statements referencing "PT", "ST" and "TT" are likewise singled out and when a symbol or template is defined by means of two such statements, these are combined.

During the expansion phase, input lines are read and matched to the template tree. When a macro call is found, the corresponding code body is examined and one of several processors is called to punch, print, move information to the input, etc. One of these processors is an interpreter for the subset of Snobol which makes up the general code body statement. If the statement cannot be recognized as a special case, it is handled by this interpreter. Each such statement has been converted into a list with sublists for the pattern and replacement, each of which is a list of elements. The go-to fields are replaced by links to the lists for the correct statement. If no go-to is specified, both S and F links point to the next sequential rule. Thus every rule effectively has go-to fields, and any rule which is not accessible from some other rule is not attached to the list structure. Such inaccessible rules are eliminated during garbage collection.

Limp enters the definition phase immediately after processing the flag line, and accepts definitions until a blank line is encountered where a template is expected. The expansion phase is entered with the succeeding line. (Notice that this allows for a blank line to appear anywhere within a code body.) The expansion phase continues until one of the three statements "TT = IN /", "PU = IN /" or "PR = IN /" is executed in a code body. The first statement initiates a re-entry to the definition phase. When another blank line is found, the program returns to the statement following "TT = IN /". The two statements "PU = IN /" and "PR = IN /" initiate block copying from the input unit to the punch and printer, respectively. In each case the template-matching process is bypassed. When a blank line is encountered, the processor returns to the statement following the one which initiated the copying.

The major drawback of the current implementation is that Wisp stores a single character per word, which increases both the storage requirements and execution times of Limp. When it is processing macros with many statements which are not special cases, the name "Limp" seems particularly apt. This defect of Wisp is being remedied by the inclusion of packed character strings as data items.

The major extension contemplated for Limp is an alteration of the template scan. It is proposed that, instead of a single parameter flag, we allow named parameters of the form *NAME*. Such a parameter would be defined by a Backus Normal Form expression, and the scanner would have the power of a syntax-directed compiler: A substring of the input line would only match a named parameter if it had the specified syntax. Each BNF rule would have an associated code body, which would be called by a function EX(d0). The value of this function would be the string produced by the code body of the rule used to recognize parameter *d*. A parameter with no name (i.e., "**") would be handled in the same way as the current version handles all parameters. The extended Limp would be closely related to the syntax-directed compiler described by Warshall and Shapiro [12].

Extending Limp in the manner described above would result in a program with considerable power, but which could be used for simple things. One of the drawbacks of a syntax-directed compiler is that one must specify the entire syntax of a language. This would *not* be the case in extended Limp. One could make small changes in the apparent behavior of an existing compiler by defining small extensions or changes in its syntax in Limp—the bulk of the syntax analysis would still be done by the existing compiler.

A macro processor which has this ability to make small changes in an existing compiler was recently described by Leavenworth [13]. His program uses syntactic information during the recognition process and allows fixed strings to be interspersed with the formal parameters. However, the macro name must precede the first parameter and must be unique. Apparently recognition of the macro is performed on the basis of the name alone, and syntactic information is used solely to establish the values of the actual parameters. The code body of the macro allows for conditional generation in a rather restricted manner. By including extra information in the template (*following* the name) one macro definition essentially becomes several. The code body then has ways of generating appropriate code for each alternative.

6. Summary

In this paper we have described an approach to language-independent macro processing. The key principle of LIMP is string manipulation with parameter substitution, a technique which has recently been used in two similar processors: Strachey's General Purpose Macrogenerator [3] and Mooers' TRAC [5]. These systems are very close to LIMP in design goals, but their realization displays a fundamental difference in programming philosophy. Both employ the notion of "nested functional expressions" in which the macro is written as a composite function with strings as arguments. LIMP treats the macro as a procedure to be executed rather than a function to be evaluated.

LIMP has sufficient capacity to provide the programmer with a powerful tool for modifying the apparent behavior of an existing compiler, without the necessity for a complete redefinition of that compiler's language. This program has been available for general use in the Basser Computing Department since April, 1966. Typical applications have been to reduce the amount of punching required in ALGOL programs, to eliminate the need for many of the line declarations in a flowchart language, and to avoid the individual specification of large numbers of constants in plotting programs written in assembly code.

Our experience with LIMP has led us to consider the possibilities of incorporating most of its features into a general text-editing system, thus combining the tasks of text correction and expansion. Research in this area is being carried out in connection with the development of a multicomputer network by the staff of the Basser Computing Department.

Acknowledgment. The general approach taken by LIMP is similar to that of most other macro processors—its relation to BEFAP and IBMAP can easily be seen. Many features of LIMP are a direct outgrowth of the author's work on several WISP compilers, in collaboration with Prof. M. V. Wilkes (Univ. of Cambridge), H. Schorr (IBM), and R. J. Orgass (Yale Univ.). Special thanks are due to Dr. M. H. Rathgeber of the University of Sydney for his aid in providing test cases and suggestions for both the processor and its documentation, and to the two referees who struggled through the first version of this paper and provided an all-important fresh outlook.

RECEIVED MAY, 1966; REVISED NOVEMBER, 1967

REFERENCES

1. MCILROY, M. D. Macro instruction extensions of compiler languages. *Comm. ACM 3* (April, 1960), 214.
2. HALPERN, M. I. XPOP: a metalanguage without metaphysics. Proc. AFIPS 1964 Fall Joint Comput. Conf., Vol. 26, p. 57.
3. STRACHEY, C. A general purpose macrogenerator. *Comput. J. 8* (Oct. 1965), 225.
4. GRAHAM, M. L., AND INGERMAN, P. Z. An assembly language for reprogramming. *Comm. ACM 8* (Dec. 1965), 782.
5. MOOERS, C. N. TRAC, a procedure-describing language for the reactive typewriter. *Comm. ACM 9* (March, 1966), 215.
6. FARBER, D. J., GRISWOLD, R. E., AND POLONSKY, I. P. SNOBOL, a string manipulation language. *J. ACM 11* (Jan. 1964), 21.
7. ——. The SNOBOL3 programming language. *BSTJ 65* (July–Aug. 1966), p. 895.
8. WAITE, W. M. A language-independent macro processor. Basser Computing Dept. Tech. Report 41, Sydney, Australia, March, 1966.
9. GRISWOLD, R. E., AND POLONSKY, I. P. String pattern matching in the programming language SNOBOL. Bell Laboratories, July 1, 1963.
10. WILKES, M. V. An experiment with a self-compiling compiler for a simple list processing language. In Richard Goodman (ED.), *Annual Review in Automatic Programming, Vol. 4*, Pergamon Press, New York, 1964, p. 1.
11. ORGASS, R. J., SCHORR, H., WAITE, W. M., AND WILKES, M. V. WISP—a self-compiling list processing language. Basser Computing Dept. Tech. Report 36, Sydney, Australia, Oct. 1965.
12. WARSHALL, S., AND SHAPIRO, R. M. A general-purpose table-driven compiler. Proc. AFIPS 1964 Spring Joint Comput. Conf., Vol. 25, p. 59.
13. LEAVENWORTH, B. M. Syntax macros and extended translation. *Comm. ACM 9* (Nov. 1966), 790.

Proposed USA Standard

COBOL

is now available

- This important Proposed USA Standard COBOL is contained in an issue of the ACM SICPLAN Notices (Volume 2, Number 4, April 1967), which was distributed in June to the regular ACM SICPLAN mailing list. It has also been sent to the COBOL Information Bulletin mailing list.
- To interested persons not on either of the mailing lists for the above publications, this Proposed COBOL Standard is available at $3.00 per copy. Orders must be prepaid and should be addressed: COBOL, Association for Computing Machinery, 211 East 43rd Street, New York, New York 10017. A special price of $2.50 per copy is being granted *by ACM* for bulk orders of 50 or more.
- USA Standards Committee X3 has authorized publication of this document, which contains 538 pages, to elicit comment and criticism from the data processing community prior to voting on its acceptance as a USA Standard. Comments should be addressed to: X3 Secretary, Business Equipment Manufacturers Association, 235 East 42nd Street, New York, New York 10017.

REFERENCES

1. Gray, J. C. Compound data structure for computer aided design; a survey. Proc. ACM 22nd. Nat. Conf. 1967, Thompson Book Co., Washington, D. C., pp. 355-365.
2. Hansen, W. J. The impact of storage management on the implementation of plex processing languages. Tech. Rep. No. 113, Computer Science Dep., Stanford U., Stanford, Calif., 1969.
3. Knuth, D. E. *The Art of Computer Programming, Vol. 1.* Addison-Wesley, Menlo Park, Calif., 1968.
4. Lang, C. A., and Gray, J. C. ASP—A ring implemented associative structure package. *Comm. ACM 11*, 8 (Aug. 1968), 550-555.
5. McCarthy, J., et al. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.
6. Minsky, M. L. A LISP garbage collector using serial secondary storage. MIT Artificial Intelligence Memo. No. 58, MIT, Cambridge, Mass., Oct. 1963.
7. Ross, D. T. A generalized technique for symbol manipulation and numerical calculation. *Comm. ACM 4*, 3 (Mar. 1961), 147-150.
8. ——. The AED free storage package. *Comm. ACM 10*, 8 (Aug. 1967), 481-492.
9. Reynolds, J. C. Cogent programming manual. Argonne Nat. Lab. Rep. No. ANL-7022, Argonne, Illinois, Mar. 1965.
10. Schorr, H., and Waite, W. M. An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM 10*, 8 (Aug. 1967), 501-506.
11. Stygar, P. LISP 2 garbage collector specifications. TM-3417/500/00, System Development Corp., Santa Monica, Calif., Apr. 1967.
12. Wiseman, N. E. A simple list processing package for the PDP-7. In *DECUS Second European Seminar*, Aachen, Germany, Oct. 1966, pp. 37-42.

A Base for a Mobile Programming System

Richard J. Orgass
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

and

William M. Waite
University of Colorado, Boulder, Colorado

An algorithm for a macro processor which has been used as the base of an implementation, by bootstrapping, of processors for programming languages is described. This algorithm can be easily implemented on contemporary computing machines. Experience with programming languages whose implementation is based on this algorithm indicates that such a language can be transferred to a new machine in less than one man-week without using the old machine.

KEY WORDS AND PHRASES: bootstrapping, macro processing, machine independence, programming languages, implementation techniques
CR CATEGORIES: 4.12, 4.22

1. Introduction

The development of many special purpose programming languages has increased the overhead associated with changing computing machines and with transferring a programming language from one computer center to another. A number of writers have proposed that these languages should be implemented by bootstrapping. (Since this work is well known, it is not summarized in this introduction.)

The description is given of a bootstrapping procedure which does not require a running processor on another machine for its implementation. A compiler is described which can be trivially implemented "by hand" on contemporary computing machines. This simple compiler is then used to translate a more elaborate one, and so forth, until the desired level of complexity is reached.

This paper is a complete description of the first stage of such a bootstrapping procedure. It is organized as follows: in Section 2, the processing performed by the simple compiler (SIMCMP); in Section 3, some examples of its use; in Section 4, the environment which must be coded for a particular computing machine; and in Section 5, the SIMCMP algorithm.

2. Specifications for SIMCMP

The algorithm described in this paper was constructed to provide a compiler of minimum length and complexity which is adequate to serve as the base for the implementation, by bootstrapping, of programming systems. SIMCMP is essentially a very simple macro processor. Source language statements are defined in terms of their object code translations. These definitions are stored in a table. A source program is translated one line at a time. The input line is compared with all of the entries in the table. When a match occurs, the object code translation of the input line, with appropriate parameter substitutions, is punched. SIMCMP can be used to translate any source language which can be defined by means of simple substitution macros with single character parameters.

Several terms as they are used in this paper are illustrated here. In a conventional macro processor, a *macro definition* is of the form:

MACRO NAME ($P_1, \cdots, P_n$)
Code body
END

The strings 'MACRO' and 'END' serve to delimit the macro definition. 'NAME' stands for the *name of the macro*, and 'P_1', $\cdots$, 'P_n' stands for the *formal parameters* of the macro. This macro is *called* by a line of the form 'NAME ($A_1, \cdots, A_m$)' where m is less than or equal to n. When this *call* is encountered in the program text, the code body of the macro NAME is *evaluated* by substituting the values of 'A_1', ⋯ , 'A_m' for occurrences of 'P_1', ⋯ , 'P_n'. If m is less than n, then values for *actual parameters* 'A_{m+1}', ⋯ , 'A_n' are generated by the macro processor in some regular way.

SIMCMP differs from a conventional macro processor in that the line which introduces a macro definition may be an arbitrary string of characters, with parameters indicated by a special symbol called a *parameter flag*. This line is referred to as the *template*. The effect of using a template is to allow the name part of a traditional macro to be replaced by a "distributed name" which consists of all the characters in the line except the parameter flags. Except for the parameter flags, each character of the template must match the corresponding character of the input line exactly. A parameter flag, however, may match any single character. That is, the template line may be thought of as a mask consisting of literal characters interspersed with "holes" which correspond to arbitrary characters in the input string being matched. The character which is matched by a particular parameter flag becomes the value of the formal parameter which is denoted by this parameter flag.

The code body of a SIMCMP macro consists of lines in the target language with references to parameters substituted for some elements of the lines. That is, each line consists of a series of strings of characters and references to parameters. A reference to a parameter is signaled by a special character called a *machine language* (MCT) *parameter flag*. This flag is followed by two digits, the first of which specifies the number of the formal parameter in the template, counting from the left. The second digit of the parameter call defines the type of conversion to be used in the particular substitution instance of the formal parameter. A maximum of nine formal parameters, numbered 1 to 9, may be specified in a single template. The two conversion types which are available are described below.

The SIMCMP translator has two main phases: macro definition and macro expansion. During the macro definition phase, user-defined macros are read and stored in an array. The first input line for SIMCMP is called the *flag line* and contains five control characters. The first is the *source language end-of-line flag*, a character which marks the end of characters to be translated in an input line. The second character is the *source language parameter flag* which is used in templates to indicate the position of formal parameters. The third and fourth characters of the flag line, respectively, are the MCT *end-of-line* and the MCT *parameter flags*. The first marks the end of useful information in a code body line; the second indicates that a converted actual parameter should be inserted at this point in the machine code output. The fifth character on the flag line must be the character '0'. Following the flag line is a series of macro definitions. Each macro definition is terminated by a line whose first character is the MCT end-of-line flag. After this line, a new template must appear. If a particular macro is the last one to be defined, then the first *two* characters of its terminating line are machine code end-of-line flags.

After all of the macro definitions have been read, SIMCMP enters the expansion phase. A single source statement is read into the array. Reading is terminated when a source end-of-line flag occurs in the input stream. This statement is translated by successively matching it against each of the defined templates. Before this comparison is done, a check is made to ensure that the source line is not void (a void line terminates the program to be translated). During the matching process, a character which is matched against a source language parameter flag in the template is placed in the corresponding parameter storage word. If all the characters in the input line are successfully matched, the input line is accepted as a call on the macro whose template is currently being scanned. If SIMCMP fails to find a match for some character of the input line, an attempt is made to match this input line to the next template. If all templates have been compared with an input line and there is no match, then the input line is assumed to be in the target language and it is punched out without translation. After processing an input line, the next input line is processed, and so forth, until a viod line is encountered.

Since the templates are scanned in order, an input line which matches several templates will be treated as an instance of the first template encountered. Therefore, it is the responsibility of the user to ensure that the ordering of the macro definitions will not produce strange results.

When an input line has been matched to a particular template, the lines of its associated code body are punched out. Characters other than the formal parameters are punched exactly as they appear in the code body and call on formal parameters are replaced by the values specified by the conversion digit and the actual parameter value. Two conversion types are available: type 0 and type 1. Type 0 conversion is a direct copy of the actual parameter into the output string. For each computing machine, the user of SIMCMP must define a one-to-one mapping, M, from the character set of his computing machine onto a set of positive integers. The only restriction on M is that the characters 0 to 9 must map onto successive integers. Type 1 conversion produces as output the numeral which denotes the image of the actual parameter under the mapping M.

For some target languages, it is necessary to SIMCMP to be able to generate arbitrary, unique symbols. This capability is provided by means of parameter zero. Up to ten unique symbols may be generated during the expansion of a single macro by referring to parameter zero with conversion type 0 to 9. The conversion type, in this case, indicates which of the created symbols is being referred to (for example, '00' refers to the first created symbol, '01' refers to the second created symbol, etc.). These symbols are three-digit decimal numerals; the first one denotes the integer 100. These numerals are supplied by a "location counter" which is maintained by SIMCMP. This location counter is initially set to the value 100; after processing a macro which contains references to parameter

0, its value is incremented by one more than the highest conversion digit used.

The structure of the translation performed by SIMCMP imposes restrictions on both the source language and the target language. Restrictions on the source language syntax are not important because the only purpose of SIMCMP is to compile the next compiler in the bootstrap sequence. Restrictions on the target language are more serious; so we have attempted to generalize the program without compromising our design objectives.

3. Examples

The compiler for the list processing language WISP [1] is written in a subset of itself which can be compiled by SIMCMP. SIMCMP has been used to implement the WISP language for several computing machines. The programming effort required to implement WISP in this manner is about one man-week. The macro processor LIMP [2] is written in the programming language WISP and, consequently, is available once WISP has been implemented.

WISP may be described approximately by saying that it is a simple version of the "program feature" of LISP [3]. The examples given here are taken from the subset of WISP which is compiled by SIMCMP. In order to provide examples which do not refer to a particular computing machine, the FORTRAN II programming language is used as the target language. In actual practice, the target language would be the assembly code of a particular machine.

Figure 1 contains examples of two SIMCMP macros and illustrations of the output produced by SIMCMP for particular source language strings as inputs. In this figure, the source language and MCT end-of-line flags are "·". The source language parameter flag is "*" and the MCT parameter is "'".

An example of the use of parameter 0 is given in Figure 2. The flags in this figure are the same as those in Figure 1. Since the target language is FORTRAN II, arbitrary labels must be generated because three labels must be specified for the IF statement. When the condition is not satisfied, control is to pass to the next statement, which must be assigned a label. SIMCMP produces this label automatically by means of parameter 0.

4. The Environment for the Simple Compiler

The environment for SIMCMP is defined to be all programs which are required to use SIMCMP on a particular computing machine. That is, the environment includes input and output programs, the assembler for this computing machine, and a driver program. (The SIMCMP algorithm is given as a procedure.) The input program is used to read from a source created by the user. This source is to be organized into lines. For remote terminal or paper tape oriented machines, a line is delimited by carriage returns. For card oriented machines, each card is a line and is considered to have a carriage return following the 80th character.

The output program is used to write text which is to be processed by the assembler; the output text is also organized into lines.

The input program (IREAD) is an integer procedure with one parameter. The value returned by this procedure depends on the value of the parameter and the next character in the source. The parameter of IREAD may have the value 0 or 1. If IREAD is called with a parameter whose value is 1, then the value returned by

```
* = CAR.*.
      I = CDR('21).
      CDR('11) = CAR(I).
.X
```

(a) Example of SIMCMP macro definition.

```
A = CAR B.
```

(b) A string which matches the template of (a).

```
I = CDR(38)
CDR(36) = CAR(I)
```

(c) A possible output from macro (a) when string (b) is the input

```
*** * = *** *.
        I = CDR('41).
        J = CDR('81).
        '10'20'30(I) = '50'60'70(J).
  .X
```

(d) Another example of a SIMCMP macro definition

```
CAR A = CDR B.
```

(e) A string which matches the template of (d).

```
I = CDR(36)
J = CDR(38)
CAR(I) = CDR(J)
```

(f) Output produced by SIMCMP when string (d) is the input.

FIG. 1. Examples of SIMCMP macros

```
TO ** IF CAR *!= CDR *.
      I = CDR('31).
      J = CDR('41).
      (IF (CAR(I) - CDR(J))'00, '10'20, '00.
'00   CONTINUE.
.X
```

(a) A SIMCMP macro definition which uses parameter zero.

```
TO 13 IF CAR A = CDR B.
```

(b) A string which matches the template of (a).

```
      I = CDR(36)
      J = CDR(38)
      IF (CAR(I) - CDR(J))100, 13, 100
100   CONTINUE
```

(c) Output produced by SIMCMP if string (b) is the first input string which matches a template whose definition uses parameter zero.

```
TO 14 IF CAR B = CDR A.
```

(d) A string which matches the template of (a).

```
      I = CDR(38)
      J = CDR(36)
      IF (CAR(I) - CDR(J))101, 14, 101
101   CONTINUE
```

(e) Output produced by SIMCMP if string (d) is the second input string which matches a template whose definition uses parameter zero.

FIG. 2. Example of the use of parameter zero

the procedure is the integer which corresponds to the next character in the source under the mapping M. Note that it is assumed that the source is organized so that the first character of a line immediately follows the carriage return of the line which precedes it. The carriage return itself corresponds to the integer −1.

If IREAD is called with a parameter whose value is 0, then the value returned by the procedure is undefined.

```
      SUBROUTINE SIMCMP(LIST,KMAX)
      DIMENSION LIST(KMAX)
C     READ CONTROL CHARACTERS, IN THE ORDER -
C                SOURCE EOL, SOURCE PARAM, MCT EOL, MCT PARAM, ZERO.
      DO 1 I=1,5
    1 LIST(I)=IREAD(1)
      LIST(6)=100
      K=17
C     READ MACRO DEFINITIONS
    2 I=IREAD(0)
      L=K+1
      LIST(L)=-1
      I=L
    3 IF (I .GE. KMAX) STOP
      I=I+1
      LIST(I)=IREAD(1)
      IF (LIST(I) .NE. LIST(1)) GO TO 3
    6 J=IREAD(0)
      J=I+1
    7 I=I+1
      IF (I .GE. KMAX) STOP
      LIST(I)=IREAD(1)
      IF (LIST(I) .NE. LIST(4))  GO TO 12
      LIST(I)=LIST(5)-IREAD(1)-7
      I=I+1
      LIST(I)=IREAD(1)-LIST(5)
      IF (LIST(I-1) .NE. (-7))  GO TO 7
      IF (LIST(L) .LT. LIST(I))  LIST(L)=LIST(I)
      GO TO 7
   12 IF (LIST(I) .NE. LIST(3))  GO TO 7
      LIST(I)=-1
      IF (I .NE% J)  GO TO 6
      LIST(K)=I
      K=I
      IF (IREAD(1) .NE. LIST(3))  GO TO 2
C     READ A SOURCE STATEMENT
   20 I=IREAD(0)
   21 DO 22 I=K,KMAX
      LIST(I)=IREAD(1)
      IF (LIST(I) .EQ. LIST(1))  GO TO 30
      IF (LIST(I) .EQ. (-1))  GO TO 52
   22 CONTINUE
      STOP
C     TRANSLATE ONE STATEMENT
   30 IF (I .EQ. K)  RETURN
      M=17
   31 L=8
      N=M+1
      DO 34 J=K,I
      N=N+1
      IF (LIST(N) .EQ. LIST(2))  GO TO 33
      IF (LIST(N) .EQ. LIST(J))  GO TO 34
   32 M=LIST(M)
      IF(M-K)31,50,50
   33 IF (I .EQ. J)  GO TO 32
      LIST(L)=LIST(J)
      L=L+1
   34 CONTINUE
      GO TO 41
C     PUNCH MACHINE CODE TRANSLATION
   40 CALL IPNCH(LIST(N))
   41 N=N+1
      IF (LIST(M) .EQ. N)  GO TO 47
      IF (LIST(N) .GE. (-1))  GO TO 40
      L=-LIST(N)
      N=N+1
      IF (L .EQ. 7)  GO TO 42
      IF (LIST(N) .NE. 0)  GO TO 43
      CALL IPNCH(LIST(L))
      GO TO 41
   47 LIST(6)=LIST(M+1)+LIST(6)+1
      GO TO 20
C     CONVERT A PARAMETER TO AN INTEGER
   42 LIST(7)=LIST(N)+LIST(6)
   43 I=LIST(L)
      DO 44 J=K,KMAX
      L=I/10
      LIST(J)=I-(L*10)
      IF (L .EQ. 0)  GO TO 45
   44 I=L
      STOP
   45 CALL IPNCH(LIST(J)+LIST(5))
      J=J-1
      IF (J .GE. K)  GO TO 45
      GO TO 41
C     PUNCH OUT AN UNRECOGNIZED LINE AFTER GETTING IT ALL IN
   50 J=I+1
      DO 51 I=J,KMAX
      LIST(I)=IREAD(1)
      IF (LIST(I) .EQ. (-1))  GO TO 52
   51 CONTINUE
      STOP
   52 DO 53 J=K,I
   53 CALL IPNCH(LIST(J))
      GO TO 21
      END
```

Fig. 3. The SIMCMP algorithm

However, this call causes the input stream to be moved forward until a carriage return character has been passed. That is, after such a call, the next call to IREAD with a parameter whose value is 1, returns as its value the integer which corresponds to the first character of the next input line. Successive calls IREAD(0) may be used to skip input lines.

The output program (IPNCH) is a procedure with one argument which is used to convert integers to the characters to which they correspond. The statement CALL IPNCH(I) causes the character which corresponds to I to be placed in the output character stream. Since the integer −1 corresponds to the carriage return, CALL IPNCH(−1) terminates the current output line. That is, for a card oriented machine it causes the termination of the current output card; for a remote terminal or paper tape oriented machine, it causes the carriage return character to be placed in the output stream.

By contemporary standards, the assembler required for the SIMCMP environment is quite simple. This assembler must be capable of assigning values to symbolic addresses which are strings built up out of numerals or numerals and other characters. The symbols generated by SIMCMP are always strings of numerals. However, the translation rules may be written so that a string of numerals is preceded or followed by another string of characters. This assembler must have a facility for reserving storage.

The driver program defines the storage used by the procedure SIMCMP and calls this procedure. It may also be used to take care of bookkeeping functions. During translation, SIMCMP requires a single integer array and six temporary storage locations. The array must be large enough to contain all of the translation rules (stored one character per element) and a single input line. In addition, fifteen elements of the array plus two elements per translation rule are used for control information.

5. The SIMCMP Algorithm

The SIMCMP algorithm is shown in Figure 3. This algorithm is stated as a FORTRAN IV subroutine. It has been used in this form with an IBM 7044, an IBM 360/50, and a CDC 6400.

The two parameters are the array described in Section 4 (LIST) and the number of elements in this array (KMAX). The comments in the program explain its operation, although this information is not required for the use of the program.

RECEIVED DECEMBER 1967; REVISED MAY 1969

REFERENCES

1. ORGASS, R. J., SCHORR, H., WAITE, W. M., AND WILKES, M. V. WISP—A self-compiling list processing language. Tech. Rep. No. 36, Basser Computing Dep., U. of Sydney, Australia, Oct. 1965. (Also reprinted by the Dep. of Electrical Engineering, U. of Colorado, Boulder, Col., and by the Columbia U. Computer Center, New York.)
2. WAITE, W. M. A language independent macro processor *Comm. ACM 10*, 7 (July 1967), 433–440.
3. MCCARTHY, J. ET AL. LISP 1.5 programmer's manual. MIT Computation Center, Cambridge, Mass., 1962.

Syntax Macros and Extended Translation

B. M. Leavenworth
*International Business Machines Corporation,**
Yorktown Heights, New York

A translation approach is described which allows one to extend the syntax and semantics of a given high-level base language by the use of a new formalism called a syntax-macro. Syntax-macros define string transformations based on syntactic elements of the base language. Two types of macros are discussed, and examples are given of their use. The conditional generation of macros based on options and alternatives recognized by the scan are also described.

Introduction

The procedure concept in programming has been developed into an elegant and powerful tool [1-3]. However, the potential extension of current procedural languages into special purpose areas seems to require a more flexible facility than offered by the procedure. On the other hand, the conventional macro, also considerably developed [4, 5], has still been based essentially on symbolic assembly code. The purpose of this paper is to suggest a generalization of the macro concept to high-level languages, which allows the programmer to extend the syntax and semantics of a given base language by new statements or expressions. The constituents of the new type of macro, called a syntax-macro, are syntactic elements, or constructs of the base language, rather than fragments of machine instructions as is the case with conventional macros. The objective of this work is to develop powerful modes of expression and reference by using flexible syntax and multiple levels of definition.

Syntax Macros

A syntax-macro has two parts:

(1) a macro *structure* which describes the syntax of the source text to be recognized;

(2) a macro *definition* which describes the semantics of the corresponding macro structure.

The macro structure is a string of metavariables and basic symbols. The first symbol of the string must be a unique basic symbol called the *macro delimiter*. The metavariables are the parameters of the macro and are drawn from the metavariables (i.e., constituents of the syntax) available in the base language. For the present discussion, two types of macros are assumed to be available: statement macros and function macros.

The macro definition is also a string of metavariables and basic symbols. Each metavariable in the definition must appear in the structure. Metavariables in the structure are ordered from left to right, and each is referenced in the definition by inserting the ordinal number of the variable in question, preceded by the metasymbol, '$'. A syntax-macro may be thought of as a string function whose domain is the macro structure and whose range is the macro definition. Both structure and definition must have the same syntactic type. A syntax-macro is declared as follows:

macro *U* **define** *V* **endmacro**

where macro stands for either **smacro** (statement macro) or **fmacro** (function macro), *U* stands for the macro structure, and *V* for the definition.

To illustrate a syntax-macro, it is necessary first to specify a base language. The following grammar defines a base language L_B :

```
program ::= block
block ::= begin [local identifier;] ··· state-list end
state-list ::= statement [; statement] ···
statement ::= variable ← express | go to identifier | if express
  then  statement | block | result  express | label  statement |
  s-macro-call
express ::= a-term [relatop a-term]
a-term ::= b-term [{+ | −} b-term] ···
b-term ::= primary [{* | /} primary] ···
primary  ::=  variable | constant |(express)| block | f-macro-call
variable ::= identifier [(express[, express] ··· )]
relatop ::= < | ≦ | = | ≠ | > | ≧
```

The syntax notation employed here follows that in the PL/I report [3], namely:

{ } braces denote grouping,
[ ] square brackets denote optional occurrence,
| vertical stroke separates alternatives,
··· three dots denote the occurrence of the immediately preceding syntactical unit one or more times in succession.

Metavariables are lower-case words, with the exception of the following: "identifier," "constant" and "label"

---

* Advanced Systems Development Division

(which has the form "identifier:") are scanned by the translator as basic symbols. Other basic symbols are special characters and keywords (denoted by boldface type).

The semantics of L_B are not defined rigorously here, but they could be specified by a real or simulated machine. Note that a block may act as either a statement or a primary. The use of a block as a primary allows the designated result of a series of statements (status changes) to be embedded as a value in the context of a larger expression. This end value is defined by using the **result** statement before exit from the block. A block also serves to qualify local identifiers and labels, and to protect them from similar names in the external environment. A macro-call is a macro structure with the metavariables replaced with literal strings (actual parameters). When the macro-call is scanned, each literal string must agree with the syntactic type of its corresponding metavariable. After the proper substitutions have been made for the metavariables in the definition, the resulting string must be a syntactically correct string in the base language. A simple **for** statement may now be defined in terms of L_B by the following statement macro.

```
smacro for variable ← express to express do statement
define begin $1 ← $2;
  L1:  if $1 ≦ $3 then
        begin $4;  $1 ← $1 + 1;
          go to L1
        end
      end
endmacro
```

Notice that both the structure and definition of the above macro have the same syntactic type, i.e., statement. This is essential for correct analysis. The macro delimiter in this example is "**for**"; once a macro delimiter appears in a new macro declaration, it becomes a "reserved word" and may only be used in the call of this macro.

In the case of the macro call,

```
for k ← 1 to n+1 do j ← j+m[k]
```

the correspondence between metavariables and literal strings is shown below:

metavariable	*syntactic type*	*literal string*
$1	variable	k
$2	express	1
$3	express	n+1
$4	statement	j ← j+m[k]

The above macro call would expand into the following string:

```
begin k ← 1;
  L1:  if k ≦ n+1 then
     begin j ← j+m[k];
       k ← k+1;
       go to L1
     end
end
```

The second type of macro to be considered is the function macro. Consider the following declaration, which contains a call to the **for** macro in its definition:

```
fmacro sum express with variable ← express to express
define begin local t;
          t ← 0;  for $2 ← $3 to $4 do
            t ← t + $1;  result t
        end
endmacro
```

The purpose of the function macro is to produce a value which may be embedded in an expression. The syntactic type of an *f*-macro-call is a primary, which is consistent with the usage of ordinary functions in programming languages.

To illustrate these ideas, consider the statement,

```
c ← a * sum b(k) with k ← 1 to 10
```

which contains a call to the *f*-macro **sum**. Expansion produces

```
c ← a * begin local t;  t ← 0;
  for k ← 1 to 10 do
    t ← t + b(k);  result t
  end
```

and expansion of the **for** macro yields

```
c ← a * begin local t;  t ← 0;
  begin k ← 1;
    L1:  if k ≦ 10 then
      begin t ← t + b(k);
        k ← k + 1;  go to L1
      end
  end;  result t
end
```

Nested calls such as the following can also be made:

```
sum sum b(j, k) with k ← 1 to 20 with j ← 1 to 10
```

The effect of macro recognition and expansion is the replacement of one string by another. If the new string contains no macro delimiters, i.e., it is a string in the base language, it can then be scanned and transformed according to the semantics of the base language; otherwise the process is repeated. Because the scope of literal strings is determined by syntactic analysis, actual parameters need not be delimited by special symbols like commas in an argument list. Further, certain symbols used as separators may also be contained in actual parameters. For example, in the macro structure:

```
signal express (state-list)
```

any literal string corresponding to the metavariable "express" may contain nested parentheses.

Parsing Method

In principle, any method of syntactic analysis that produces a structural description [6] of source text can be used for macro recognition. Whenever a programmer is given the right to add new syntax rules to a base language, the parsing method should be specified in order to remove ambiguity. The method assumed in these illustrations is a top-down analysis such as that described by Schorre [7] or Conway [8]. As soon as a marco delimiter is recognized, the translator goes into a special macro scanning mode in which the usual semantic operations are suppressed, and a literal string is generated for each metavariable in the structure. At the conclusion of macro recognition, the source text is modified according to the macro definition, and the resulting text *T* replaces the original text in the source string; scanning now proceeds again starting at the beginning of *T*. Some advantages of the parsing method and syntax notation adopted here are: (1) syntax is recognition-oriented; (2) options are easy to specify; and (3) ambiguity is eliminated by virtue of the factoring or no-backup principle.

There is a possible danger of conflict of identifiers after the macro has been expanded, for example if any of the expressions used in a call on the **sum** function would refer to a free variable named *t*. This conflict is avoided if identifiers and labels are qualified by their block numbers in an internal representation while the macros are being scanned.

Conditional Macro Generation

Macro structures may contain options or alternatives which may be used for conditional macro generation. As an example, consider a slightly more complicated **for** statement:

smacro for variable ← express {while express^1 |[by express^2] to
express^3}
do statement

An option or alternative is called a group; groups are numbered from left to right as shown; i.e., a new number occurs just before each |,], and } character. Metavariables are also numbered from left to right.

Associated with each group in the macro structure is a corresponding group in the definition. Each definition group is delimited by braces, and is referenced by the ordinal number of the corresponding structure group. Definition groups may appear in any order, with repetition allowed.

If a group is recognized during a scan of the macro structure, its corresponding definition group is scanned; otherwise the group is bypassed. One possible macro definition of the given **for** statement is shown below

```
define begin L1:  $1 ← $2;
             L2:  if {1 $3 then begin $6;  go to L1}
                  {3 $1 ≦ $5 then begin $6;
                  $1 ← $1 + {2 $4}{¬2 1};  go to L2}
                  end
       end
endmacro
```

The notation {¬*n* ···} means that if the *n*th group is *not* recognized during scan of the structure, the definition group is scanned; otherwise it is bypassed.

Implementation

A syntax-macro translator has been implemented as part of an experimental simulation language under development. The translator utilizes an interpretive language to define the syntax of the given base language.

The macro recognition and generation is performed by a small set of string manipulation routines written in FORTRAN. These routines perform the following functions:

Initialize string pool (link up string elements).
Create a string (unlink first element in pool and point to it).
Destroy a string (return string to pool).
Add value to a string (unlink first element in pool, store value in it and append to string).
Add string$_2$ to string$_1$ (concatenate string$_2$ to string$_1$ without copying).
Read next element in string (pointer advances).
Is string exhausted? (has pointer advanced to end of string?)

Concluding Remarks

The development of syntax-directed compilers has eased the task of producing scanners for different programming languages. Unfortunately, the specification of semantics has proved to be a more difficult problem. The present approach attempts to solve this problem by defining extensions in terms of semantics already provided in the base language. This avoids the respecification of syntax and semantics for the kernel of each new special language, assuming they have a common base. This approach pays off, because an underlying uniformity of structure can be shared by a large class of special purpose languages.

Other advantages of syntax-macros are:

1. Flexible format allows convenient language extension.
2. Multiple levels of definition are possible.
3. Macro parameters can be any syntactic element defined in the base language.
4. Special separators are not required for macro parameters.

While syntax-macros allow a flexible format, they still have essentially a prefix format which rules out the handling of infix operators. This limitation, which probably could be removed by a more general approach, is certainly one of the first things that should be treated in any extension.

The type of conditional macro generation described in this paper does not provide the full power of compile-time imperatives, since descriptive ability has been emphasized, rather than programming orientation. The macro facility described above should also allow repetition over lists [4] in order to increase descriptive power.

The relative ease or difficulty of performing extensions of a base language is a function of the choice of primitives in the base language. The really difficult problems in the development of programming languages are those of representation, data structuring, and storage allocation. It is clear that these problems will not be solved by syntactic methods alone. However, it is hoped that the approach outlined in this paper can be used in conjunction with other formalisms in a combined attack on the problems of improving programming languages.

Acknowledgment. The author would like to express his appreciation to K. R. Blake for many stimulating discussions on this subject. Also appreciated are valuable comments by Prof. D. E. Knuth and the referees of this paper.

RECEIVED FEBRUARY, 1966; REVISED MAY, 1966

REFERENCES

1. Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6* (Jan. 1963), 1–17.
2. WIRTH, N. A generalization of ALGOL. *Comm. ACM 6* (Sept. 1963), 517–554.
3. IBM Operating System/360. PL/I: language specifications. Form C28-6571, IBM Corp., 1966.
4. MCILROY, M. DOUGLAS. Macro extensions of compiler languages. *Comm. ACM 3* (Apr. 1960), 214–220.
5. FERGUSON, DAVID E. The evolution of the meta-assembly program. *Comm. ACM 9* (March 1966), 190–193.
6. CHOMSKY, NOAM, AND MILLER, GEORGE A. Introduction to the formal analysis of natural languages. *Handbook of Mathematical Psychology, Vol. II.* John Wiley, New York, 1963, pp. 283–306.
7. SCHORRE, D. V. Meta II. A syntax oriented compiler writing language. Proc. ACM 19th Nat. Conf., Philadelphia, Pa., Aug. 1964, ACM Publ. P-64.
8. CONWAY, MELVIN E. Design of a separable transition-diagram compiler. *Comm. ACM 6* (July 1963), 396–408.

LARSEN—cont'd from p. 789

an effective data processing technique. The physical realization of this concept has been seen to be highly modular and suitable for programming implementation.

The Data Filter behaves as a two-port data filter within an interpretive processing environment, and represents the physical realization of the data filtering concept. This is accomplished by associating format declarations with its input and output ports which define its processing characteristics. The desired data string is sequentially constructed in the OUT buffer by filtering datum in the IN or HOLD buffers through these format declarations. A Procedural Controller is employed to synchronize loading of the IN buffer and dumping of the OUT buffer to achieve specific data processing results as implied by the job being processed.

RECEIVED FEBRUARY, 1966; REVISED JUNE, 1966

REFERENCES

1. Functional design specification of a data retrieval model (ABACUS). SID 66-175-2, North American Aviation, Inc., Downey, Calif., March 1966, 156pp.
2. ABACUS user's manual. SID 66-175-1, North American Aviation, Inc., Downey, Calif., March 1966, 148pp.

Algorithms

J. G. HERRIOT, Editor

Algorithms Policy as revised in August, 1966 to include FORTRAN appears on page 823.

ALGORITHM 292
REGULAR COULOMB WAVE FUNCTIONS

WALTER GAUTSCHI (Recd. 8 Oct. 1965)
Purdue University, Lafayette, Indiana and Argonne National Laboratory, Argonne, Illinois

Work performed under the auspices of the U. S. Atomic Energy Commission.

```
real procedure t(y);  value y;  real y;
comment This procedure evaluates the inverse function t = t(y)
  of y = t ln t in the interval y ≧ −1/e, to an accuracy of about
  4 percent, or better. Except for the addition of the case
  −1/e ≦ y ≦ 0, and an error exit in case y < −1/e, the procedure
  is identical with the real procedure t of Algorithm 236;
begin real p, z;
  if y < −.36788 then go to alarm 1;
  if y ≦ 0 then t := .36788 + 1.0422 × sqrt(y + .36788) else
  if y ≦ 10 then
  begin
    p := .000057941 × y − .00176148;  p := y × p + .0208645;
    p := y × p − .129013;  p := y × p + .85777;
    t := y × p + 1.0125
  end
  else
  begin
    z := ln(y) − .775;  p := (.775−ln(z))/(1+z);
    p := 1/(1+p);  t := y × p/z
  end
end t;
procedure minimal (eta, omega, eps, la1, dm);
  value eta, omega, eps; real eta, omega, eps, la1, dm;
comment This procedure assigns the value of λ1′ to la1, accu-
  rately to within a relative error of eps, where {λL′} is the minimal
  solution (normalized by λ0′=1) of the difference equation
```

$$\lambda_{L+1} - \frac{2L+1}{L+1}\omega\lambda_L - \frac{L^2+\eta^2}{L(L+1)}\lambda_{L-1} = 0 \quad (\omega \neq 0).$$

```
  (For terminology, see [3].) If {λL} denotes the solution corre-
  sponding to initial values λ0 = 1, λ1 = ω − η, the procedure also
  assigns to dm the value λ1 − λ1′. The negative logarithm of
  |λ1 − λ1′| may be considered a measure of the "degree of mini-
  mality" of the solution {λL};
begin integer L, nu,  real eta2, r, ra;
    eta2 := eta ↑ 2;
    nu := 20;  ra := 0;
L1: r := 0;
    for L := nu step −1 until 1 do
    r := −(L ↑ 2+eta2)/(L×((2×L+1)×omega−(L+1)×r));
  if abs(r−ra) > eps × abs(r) then
  begin
    ra := r;  nu := nu + 10;  go to L1
  end;
  la1 := r;  dm := omega − eta − r
end minimal;
```

Building a mobile programming system

W. M. Waite*

* *Department of Electrical Engineering, University of Colorado, Boulder, Colorado, U.S.A.*

The techniques of abstract machine modelling and macro processing can be used to develop programs of great mobility. This paper describes the concepts and construction of a system which has been implemented on nine different computers. In no case was more than one man-week required, and most implementations went more rapidly than that.

(Received June 1969)

1. Introduction

The *mobility* of a program is a measure of the ease with which it can be implemented on a new machine. A user of a highly mobile program suffers minimum disruption of his work when his computer is upgraded or he moves to a new installation. Because his program is mobile, only a small amount of effort is required to get it running on the new machine before he can continue using it. Applications programs written in high level languages such as FORTRAN, ALGOL or COBOL have reasonable mobility, provided that the author has taken some pains to avoid local idiosyncrasies. For systems software, however, the picture is much grimmer. There have been many attempts to devise high level languages for compiler writing (Feldman and Gries, 1968) and a few to use such languages for the production of operating systems (Glaser, Couleur and Oliver, 1965). Unfortunately, none of these attempts has yet met with widespread success.

A FORTRAN user requires a large programming system, consisting of the compiler and associated run-time routines. The mobility of his programs is determined by the number of installations which support this system. One can therefore argue that the mobility of a program is completely dependent upon the mobility of the programming system on which it rests. In this paper, I shall discuss a technique which I believe is fundamental to the improvement of programming system mobility. The background and general concepts from which the technique was derived are presented in Section 2, while the remainder of the paper is concerned with a specific system which has been designed and built using this technique. Section 3 describes the basic bootstrap on which the system rests, and Section 4 discusses the implementation of the common macro processor. The advantages and disadvantages of the approach are presented in the Section 5.

2. General approach

The technique rests on the fact that it is possible to identify two components of any program: the basic operations and the algorithm which coordinates these operations. Given a particular task, it is possible to define a set of basic operations and data types needed to perform the task. These operations and data types define an *abstract machine*—a hypothetical computer which is ideally suited to this particular task. The program to perform the task is then written for this abstract machine. To run the program on a real machine, it is necessary to realise the abstract machine in some way.

The abstract machine is an embodiment of the basic operations required to perform a particular task. Theoretically, it has no connection with any real machine, but practically we must always keep a wary eye on reality when designing an abstract machine. Many abstract machines can be formulated for a given task. The trick is to choose one of the right ones. Three considerations must be kept in mind—

1. The ease and efficiency with which the algorithm for accomplishing the task can be programmed in the language of the abstract machine.
2. The ease and efficiency with which the simulation of the abstract machine can be carried out on machines available currently and in the forseeable future.
3. The tools at hand for the realisation of the abstract machine.

Balancing these three considerations is very much an engineering task. *If one is stressed at the expense of the others, there will be trouble.*

An early attempt to use the abstract machine concept was the UNCOL proposal (SHARE, 1958). UNCOL was to be a *UN*iversal *C*omputer *O*riented *L*anguage which would reduce the number of translators for n languages and m machines from $n \times m$ to $n + m$. Effectively, the UNCOL proposal created a single abstract machine. In view of the three considerations mentioned above, it is easy to see why this attempt was unsuccessful—try to design an abstract machine which comes close to satisfying condition 1 above for FORTRAN, LISP (McCarthy, 1960) and SNOBOL (Griswold, Poage and Polonsky, 1969) simultaneously! The mobile programming system circumvents this difficulty by allowing a multiplicity of abstract machines.

It does not attempt to solve the $n \times m$ translator problem. The advantage of the mobile programming system lies in the fact that it reduces the specification of an algorithm to the specification of its basic operations, thus drastically reducing the amount of effort needed to implement it.

One way to carry out the realisation of an abstract machine is to devise for it an assembly language whose statements can be expressed as macros acceptable to the real machine's assembler. This approach was suggested by McIlroy (1960) and used in the implementation of L6 (Knowlton, 1966) and SNOBOL (Griswold, *et al.*, 1969). Unfortunately, the assemblers for different machines may require quite different input formats, and their macro processors often vary widely in power. The ease of transferring a program written in this way thus depends critically on the existence of a suitable assembly language for the target machine.

Recent developments in language-independent macro processing (Strachey, 1965, Waite, 1967, Brown, 1967) suggest that it is possible to make available a common macro processor. If the assembly language statements for the abstract machine are acceptable to this macro processor, then the realisation does not depend upon the macro capabilities of the target machine, but rather on the availability of the common macro processor. An example of this approach is the implementation of WISP (Wilkes, 1964). There the WISP compiler is the common macro processor, and is itself built up by bootstrapping from simpler macro processors.

3. The bootstrap

One of the design goals of the mobile programming system was that implementation on a new machine should not require a running version on another machine. The base of the system was to be easily implemented by hand if necessary. Once this base was available, it could be used to implement a common macro processor which would handle the realisation of the various abstract machines in the system. The common macro processor itself was written in the assembly language of an abstract machine, and hence the base of the system must include a simple macro processor.

An adequate macro processor can be expressed as a 91 statement program written in a restricted form as ASA FORTRAN. This program, known as SIMCMP, is described in detail by Orgass and Waite (1967). SIMCMP is written in FORTRAN for two reasons:

1. Since FORTRAN is a widely-used language, it may be available on the target machine. This means that SIMCMP can be implemented trivially.
2. FORTRAN was originally designed to be quite close to machine code, and hence an algorithm expressed in FORTRAN is easy to translate to machine code by hand. (Translation of SIMCMP to machine code for two different machines required about 4 man-hours in each case.)

The primary criterion used in the design of SIMCMP was simplicity. Only those features considered to be absolutely necessary were incorporated. SIMCMP has only one purpose: to realise the abstract machine used for the common macro processor. This approach is not the one which has been taken by most designers of mobile systems, who prefer to assume that a working version of the common processor is already available on *some* machine. They argue that if a working version is available on machine M, then a new version can be created on machine N by the following procedure:

1. Code macros which translate the source code of the processor to the assembly language of N.
2. Expand the common processor, using the macros developed in (1) and the processor existing on M. The result is an assembly language program for N.
3. Run the program resulting from (2) on N.

I must reject this procedure on the basis of my own experience. More often than not, the machines M and N are remote from one another. Since it is virtually impossible to write the macros correctly the first time, steps (2) and (3) must be iterated and the distance between the machines makes this a slow and costly business. Also the machines often have incompatible peripherals and/or different character sets: at each iteration of (2) and (3) a tedious translation must take place.

By eliminating the need for a working version, SIMCMP avoids these problems. All work is done on one machine. The abstract languages being translated are defined using a restricted character set available on most machines (43 characters of the FORTRAN set on the IBM 026 card punch). The character set used for the real machine's assembly language is completely arbitrary. It is determined by the macro definitions, which are written specifically for that machine and translated on it. This is not true for the procedure outlined above. There, machine M must be capable of writing assembly code for machine N. If machine M cannot output all of the characters needed by the assembly language of N, a translation must take place at each iteration of steps (2) and (3) above. On the other hand, suppose that all of the work is being done on N using SIMCMP. If N cannot recognise all of the 43 characters used in the abstract language, a translation need only be done once to put the source code into an acceptable form.

4. The common macro processor

As pointed out in the previous section, SIMCMP was designed primarily for simplicity and is certainly not adequate to serve as a common processor for realising abstract machines. It is impossible to produce good code using SIMCMP because decisions cannot be made and alternate expansions provided. Because there is no iteration facility, macros with argument lists of indefinite length cannot be handled. A second processor, known as STAGE2, was therefore designed as the common processor for the system. It provides all of the features normally associated with a general purpose macro processor (McIlroy, 1960). In many respects, it is quite similar to LIMP (Waite, 1967). Its input recognition procedure is language-independent, employing the LIMP type of scanning mechanism to recognise macro calls and isolate parameters. The code body, however, differs from that of LIMP. It does not include the 'grouping' concepts nor the SNOBOL interpreter. Conditional expansion, iteration and the like are provided by *processor functions* rather than by an explicit program structure. The ability to perform different parameter conversions has been retained and extended. A complete manual describing the use of STAGE2 is available (Waite, 1968).

The design of STAGE2 required a balancing of two conflicting objectives:

1. It must be translated by SIMCMP.
2. It should be as flexible and as general as possible.

My overall philosophy dictated that (1) should be given most weight, and any clear choice had to be resolved in favour of it. Thus, STAGE2 does not provide *all* features which one would like to see in a macro processor, it is relatively slow, and it requires a fair amount of data space. Its purpose is to provide a common macro processor for realising a variety of abstract machines, and not to act as a processor for day-to-day use by applications programmers. For this latter use, we are preparing versions of ML/1 (Brown, 1967) and LIMP.

STAGE2 is written in a language called FLUB (*F*irst *L*anguage *U*nder *B*ootstrap). FLUB has 28 machine operations and 2 pseudo-operations, each of which can be expressed as a macro acceptable to SIMCMP. A complete description of the characteristics and design of the FLUB language has been given by Waite (1969). The procedure for implementing STAGE2 on a new machine, *N*, is thus:

1. Implement SIMCMP on *N*.
2. Write 28 macro definitions which translate FLUB into the assembly language of *N*.
3. Translate STAGE2, using SIMCMP and the definitions of (2), and assemble the resulting program.

Once STAGE2 is running, it is possible to rewrite the 28 macros, taking advantage of the added flexibility of STAGE2. Using the version of STAGE2 already available, an optimised version of STAGE2 can thus be produced:

4. Write 28 macro definitions which translate FLUB into the assembly language of *N*. These macros use the features made available by STAGE2.
5. Translate STAGE2, using the version of STAGE2 implemented in (1)–(3) and the definitions of (4), and assemble the resulting program.

The operations of the FLUB machine are rather simple to describe and can be coded in a straightforward manner. Unfortunately, a well-known axiom in computer programming states that it is impossible to write even the simplest code correctly the first time. The macro definitions are the 'hardware' of an abstract machine, and an incorrect macro definition is analogous to a wiring error in a real computer. No manufacturer tests a computer just off the production line by running a batch of FORTRAN jobs through it, and the implementor of an abstract machine should not be forced to attempt a similar feat. Two test programs have been developed by Mr. E. H. Henninger and Mr. R. C. Dunn to facilitate checkout of the macros. These test programs were quite tedious to construct, and are subject to most of the problems of normal hardware test programs (Bashkow, Friets and Karson, 1962).

Using the test programs, steps (2) and (4) above have two sub-steps:

2,4a Write 28 macro definitions which translate FLUB into the assembly language of *N*.

2,4b Translate, assemble and run the test programs. Correct any errors in the macros which were detected, and repeat step (b) until no errors remain.

Testing the macros in this manner simplifies and speeds the implementation considerably, because it means that the implementor need not be familiar with the inner workings of STAGE2 to be able to debug his macros. Since the test programs were made available, STAGE2 has been implemented on six different machines. Four of those implementations were done by people who were not familiar with STAGE2, but in no case was such familiarity needed. The test programs detected all of the macro coding errors, and STAGE2 ran perfectly when compiled.

I cannot overstress the importance of a comprehensive macro test program in the successful implementation of a machine independent system. Macro coding errors can be very subtle, requiring hours of debugging to trace them down if the machine independent program is the only test case. A conservative estimate based on our experience is that lack of a good test program will increase the time required to complete an implementation by a factor of five when the author of the system is available to debug the macros. When he is not available, the task is almost hopeless.

5. Advantages and disadvantages

Use of the abstract machine concept allows an impressive reduction in the amount of coding necessary to establish a processor on a new machine. In the case of SNOBOL4 (Griswold, *et al.*, 1969), for example, the compiler/interpreter contains roughly 4500 lines of code, but only approximately 100 primitives must be recoded for a new machine. The FLUB machine has 28 primitives, and STAGE2 is over 850 FLUB statements. A LISP (McCarthy, 1960) system without numeric capabilities requires just 8 primitives operating on 6 data types. Although the reduction in sheer bulk of coding is a factor in the success of this approach, the decrease in code complexity is more significant. Each primitive can be specified thoroughly, and is generally quite easy to implement.

Whenever the term 'machine independence' is mentioned, questions of efficiency are bound to arise. Of course, a program implemented as described in this paper will not be as fast and tight as a hand-coded version. Two arguments can be brought to bear, however. First, extreme efficiency of the generated code may not be an issue, as long as extreme inefficiency is avoided. An interactive BASIC (Dartmouth, 1966) system which can be up and running with less than one man-week of effort may be preferable to one which runs twice as fast but requires 10 man-months to complete. Second, it is possible to optimise the program on several levels. The source language is designed to match the problem well, and a great deal of time can be put into producing an efficient program for the abstract machine. Careful construction of the macros can result in surprisingly good code for the target machine. Once the assembly language version is available, critical parts can be rewritten at leisure to further improve the program's performance.

In addition to SIMCMP and STAGE2, a comprehensive text manipulation program and two small editors have been produced using the system described in this paper. We are currently working on interactive BASIC (Dartmouth, 1966) and SNOBOL4 (Griswold, *et al.*, 1969). A 'logic analyser' (which includes automatic flowcharting) and a paginator for producing reports are in the planning stage. The list processors WISP (Wilkes, 1964), LISP (McCarthy, 1960) and L6 (Knowlton, 1966) are being considered.

The system has proved extremely mobile in practice, having been implemented on nine different machines. In each case the total effort was less than one man-week, and most went more rapidly than that. A team of two people, one thoroughly familiar with the system and the other with the target machine, have had it running in one day.

Acknowledgements

The basic ideas for the system came from some unpublished comments by T. R. Bashkow on the duality of hardware and software. Techniques were developed through the implementation of LIMP and a number of WISP compilers, in collaboration with M. V. Wilkes (Cambridge University), H. Schorr (IBM) and R. J. Orgass (IBM). Thanks are due to R. E. Griswold, J. F. Poage and I. P. Polonsky of Bell Laboratories for their willingness to discuss the inner workings of the SNOBOL 4 compiler/interpreter. Dr. P. C. Poole of the U.K. Atomic Energy Authority and Prof. Wilkes were of material assistance in criticising several drafts of this paper. Computing time was kindly made available at different times by the following institutions: Computer Center, Columbia University; University Mathematical Laboratory, Cambridge; Graduate School Computer Center, University of Colorado; Culham Laboratory, U. K. Atomic Energy Authority.

References

Bashkow, T. R., Friets, J., and Karson, A. (1963). A Programming System for Detection and Diagnosis of Machine Malfunctions, *IEEE Trans. on Electronic Computers*, Vol. EC-12, p. 10.

Brown, P. J., (1967). The ML/1 Macro Processor, *CACM*, Vol. 10, p. 618.

Dartmouth College Computer Center (1966). *BASIC*, 3rd ed. Hanover, N.H.: Dartmouth College.

Feldman, J., and Gries, D. (1968). Translator Writing Systems, *CACM*, Vol. 11, p. 77.

Glaser, E. L., Couleur, J. F., and Oliver, G. A. (1965). System Design of a Computer for Time Sharing Applications, *AFIPS Conf. Proc.* Vol. 25, p. 197

Griswold, R. E., Poage, J. F., and Polonsky, I. P. (1969). *The SNOBOL4 Programming Language*, Englewood Cliffs, N. J.: Prentice-Hall, Inc.

Knowlton, K. C. (1966). A Programmer's Description of L6, *CACM*, Vol. 9, p. 616.

McCarthy, J. (1960). Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1, *CACM*, Vol. 3, p. 184.

McIlroy, M. D. (1960). Macro Extensions of Compiler Languages, *CACM*, Vol. 3, p. 214.

Orgass, R. J., and Waite, W. M. (1967). *A Base for a Mobile Programming System*, Yorktown Heights, N.Y.: IBM Corp. (to appear in *CACM*)

SHARE Ad-Hoc Committee on Universal Languages. (1968). The Problem of Programming Communication with Changing Machines: A Proposed Solution, *CACM*, Vol. 1, p. 12.

Strachey, C. (1965). A General Purpose Macrogenerator, *Comp. J*, Vol. 8, p. 225.

Waite, W. M. (1967). A Language Independent Macro Processor, *CACM*, Vol. 10, p. 433.

Waite, W. M. (1968). *The STAGE2 Macro Processor*, Boulder, Colorado: Department of Electrical Engineering and Graduate School Computing Center, University of Colorado.

Waite, W. M. (1969). *Building a Mobile Programming System*, Boulder, Colorado: Graduate School Computer Center, University of Colorado.

Wilkes, M. V. (1964). An Experiment with a Self-Compiling Compiler for a Simple List Processing Language, *Ann. Rev. in Automatic Programming*. Vol. 4, p. 1.

Here an unconditional statement plus a conditional expression has become a conditional statement.

Fortunately both of these situations have been ruled out by Section 4.7.5.2.

Conclusion

For centuries astronomers have given the name Algol to a star which is also called Medusa's head. The author has tried to indicate every known blemish in [2]; and he hopes that nobody will ever scrutinize any of his own writings as meticulously as he and others have examined the Algol Report.

Received January 1967; revised July 1967

REFERENCES

1. Naur, P. (Ed.) Report on the algorithmic language Algol 60. *Comm. ACM 3* (1960), 299–314.
2. Naur, P., and Woodger, M. (Eds.) Revised report on the algorithmic language Algol 60. *Comm. ACM 6* (1963), 1–20.
3. Abrahams, P. W. A final solution to the dangling else of Algol 60 and related languages. *Comm. ACM 9* (1966), 679–682.
4. Merner, J. N. Discussion question. *Comm. ACM 5* (1964), 71.
5. Knuth, D. E. On the translation of languages from left to right. *Inf. Contr. 8* (1965), 607–639.
6. Dijkstra, E. W. Letter to the editor. *Comm. ACM 4* (1961), 502–503.
7. Leavenworth, B. M. Fortran IV as a syntax language. *Comm. ACM 7* (1964), 72–80.
8. Knuth, D. E., and Merner, J. N. Algol 60 confidential. *Comm. ACM 4* (1961), 268–272.
9. Ingerman P. Z., and Merner, J. N. Suggestions on Algol 60 (Rome) issues. *Comm. ACM 6* (1963), 20–23.
10. Randell, B., and Russell, L. J. *ALGOL 60 Implementation.* Academic Press, London, 1964.
11. Naur, P. Questionnaire. *ALGOL Bulletin 14*, Regnecentralen, Copenhagen, Denmark, 1962.
12. B5500 Information processing systems reference manual. Burroughs Corp., 1964.
13. van Wijngaarden, A. Switching and programming. In H. Aiken and W. F. Main (Eds.), *Switching Theory in Space Technology*, Stanford U. Press, Stanford, 1963, pp. 275–283.

The ML/I Macro Processor

P. J. Brown
University Mathematical Laboratory, Cambridge, England

A general purpose macro processor called ML/I is described. ML/I has been implemented on the PDP-7 and I.C.T. Atlas 2 computers and is intended as a tool to allow users to extend any existing programming language by incorporating new statements and other syntactic forms of their own choosing and in their own notation. This allows a complete user-oriented language to be built up with relative ease.

Introduction

A macro is basically a means of extending an existing programming language, called the *base language*, by introducing a new syntactic unit which is describable in terms of the existing syntactic units of the base language. This paper is concerned only with macro processors which operate on text and act as a preprocessor to the compiler for the base language. Most existing macro processors have a fixed base language, usually an assembly language, and the macro processor and the base language processor are regarded as a single piece of software, as exemplified by the term "macro-assembler." One of the best known macro-assemblers is Macro Fap [1]. Most macro-assemblers require that macros expand into a series of statements; it is not possible, for instance, to use a macro to generate the address field of an instruction.

As a very elementary example of the use of a typical macro-assembler, assume that a user wished to introduce a single statement that would move an item from one storage location to another on a machine that required two instructions to achieve this. To do this he would define a macro, give it a name, MOVE, say, and define the two requisite instructions as the *replacement text* of the macro. Having defined his macro, the user could then treat MOVE as if it were an assembly language statement, and the macro processor would expand this into the two given instructions. The MOVE statements are called *macro calls* and have form:

MOVE *argument* 1, *argument* 2

In the typical macro-assembler the syntax of macros is very rigid. Each line of input text must be either a statement in the base language or a macro call. The arguments of macro calls are separated by a fixed delimiter, the comma, and the *closing delimiter* (the symbol used to indicate the end of a call) is typically either a space or the end of a line.

Several macro processors with more general properties than the basic macro-assembler have been produced. Among these are XPOP [2], Wisp [3], Limp [4], GPM [5] and Trac [6]. These have extended the basic concept in two main ways. Firstly, the user has been allowed to define his own notation for writing macro calls. This notation might, for instance, be something approaching the English Language or alternatively might be close to the language of algebra. Secondly, the macro processor has been made independent of the base language and its processor. In this case the same macro processor can act as a preprocessor to any number of different languages, and there is no restriction on the syntactic forms of the base language which may be expanded.

ML/I incorporates both of these extensions and is intended to allow the user to extend any language, using his own notation. The only restriction on notation is that macro calls must commence with the macro name. ML/I recognizes a macro by the occurrence of its name and a macro call is taken as the text from the macro name up to its closing delimiter. This contrasts with macro processors such as WISP and LIMP where macro calls are recognized by comparing each line of input text with a number of templates.

Main Features of ML/I

This paper will not attempt to give a detailed description of ML/I but rather will describe its main features. For details the reader is referred to the User's Manual [7]. Firstly, the general form of a macro will be described together with examples of its applications. ML/I allows any sequence of characters to be used for each delimiter. The macro name is regarded as delimiter number zero. Hence the user could specify TO and a semicolon as delimiters of his MOVE macro and write his calls:

MOVE *argument* 1 TO *argument* 2 ;

though this in itself could hardly be claimed as a dramatic improvement. The special feature of ML/I is that the user specifies a *delimiter structure* for each macro which makes it possible for each macro to have any number of alternative patterns of delimiters. The purpose of the delimiter structure is to define the name or names of the macro and to define for each delimiter a *successor* or set of alternative successors. A successor is the next delimiter to be searched for when scanning a call. A closing delimiter may be considered as having a null successor. As an example, consider an IF macro which has alternative forms:

(a) IF *argument* 1 = *argument* 2 THEN *argument* 3 END
or (b) IF *argument* 1 = *argument* 2 THEN *argument* 3 ELSE *argument* 4 END

In the delimiter structure of this macro each delimiter has a unique successor except for the delimiter THEN which has the alternative successors END and ELSE.

The above IF macro could be extended by allowing as the first delimiter a number of alternative relational operators to "equals." This could be done by specifying the set of relational operators as alternative successors to IF. Each relational operator would have THEN as its successor. In order that a meaning can be ascribed to different delimiter names, ML/I contains a facility for placing conditional statements, operative at macro generation time, within the replacement text of a macro. These conditional statements can be used to make the form of the code replacing a macro call dependent on the delimiters used in the call.

Delimiter structures can be designed to allow macros with an indefinitely long list of arguments. Assume, for example, that it is desired to extend further the IF macro by allowing between IF and THEN a whole series of relations connected by, say, AND or OR. This could be achieved by defining the successors of each of the relational operators to be AND, OR, or THEN. The successors of OR and AND would be the set of relational operators, thus causing the delimiter structure to loop back on itself.

ML/I allows macro calls to be nested within the arguments of other macro calls. Hence it would be quite possible to write nested calls of the IF macro. The method of searching for delimiters takes care of nested calls.

Before the more basic details of ML/I are described, a few more examples of its applications will be considered in general terms in order that the reader may get some sort of a feel for the kind of macro that is normally defined.

Example 1. It is possible to design a set of macros that would be useful for referencing individual fields in blocks of data. Thus FIELD and SET macros could be designed that would allow the user to define the fields of his data blocks and then to refer to these fields. His program might read (where, for the sake of clarity, the delimiters of the FIELD macro have been italicized):

FIELD FATHER *IS WORD* 1 ;
FIELD MOTHER *IS WORD* 3 ;
FIELD AGE *IS BITS* 6 *TO* 12 *OF WORD* 4 ;
SET X = AGE(FATHER(MOTHER(Y)));

The action of the FIELD macro would be to set up a macro definition whose name was derived from the first argument of FIELD. Thus the third call of FIELD would define a macro with name "AGE(" and with closing delimiter ")". The replacement text of this macro would generate code to reference the desired field of the designated data block.

Example 2. It is possible to write a macro of form:

LOAD *argument* ;

where the argument is a general arithmetic expression. This macro would generate code to load the value of the arithmetic expression into an accumulator. The argument could be analyzed by defining the character "(" as a macro name with a right parenthesis as its closing delimiter and specifying all the permissible arithmetic operators for the intervening delimiters, of which there could be any number.

Since ML/I is independent of the base language, it is intended that it be used as a common preprocessor to all the compilers and assemblers available at an installation, in the same way as each compiler and assembler might use a common loader. In general it is true to say that the higher level the language, the less need there is for macro facilities. However, even in the most comprehensive high-level languages, macros are useful for introducing statements specially designed for the user's particular field of application.

The above examples have illustrated the primary use of ML/I, namely to allow any existing programming language to be extended to suit a particular user's requirements. This form of extension could be carried to the level where the extended language could be considered as a language in its own right.

The efficiency of code generated by ML/I depends how well the macros are designed. There are macro-time variables and conditional facilities to help eliminate the sort of inefficiencies that occur at boundaries between macros. As regards speed, ML/I would be considerably slower than a special purpose compiler for a language. Note, however, that it is not intended to be a general purpose compiler. Instead of designing a new language and writing a compiler for it, the idea is for the user to start at the other end, as it were, and extend an existing language to meet his requirements.

The usefulness of ML/I is not confined to language extension. It can also be used for some applications in text editing. For instance, it is now being used as an aid to converting from Fortran IV to a dialect of Fortran II. Here ML/I performs the transformations of which it is capable, for instance recognizing declarations of logical variables and converting statements involving them into corresponding arithmetic statements. In those cases where it cannot perform the required transformation it places a special marker beside the Fortran IV statement.

Introductory Details of ML/I

ML/I is fed some *source text*. It performs some transformations on this text and generates some *output text*, which is, in turn, normally fed to some compiler or assembler. These transformations are specified by *constructions*, which are usually defined at the start of the source text. The most important type of construction is the macro.

The character set of ML/I will vary between implementations but it should contain the letters and numbers and some *punctuation characters*. A punctuation character is any character that is not a letter or number. "Space" or "blank" is treated as a punctuation character and it is usually convenient to assume there is a character called "newline" at the end of each line of text. (This character physically exists if input is from paper tape. In the case of card input it would have to be specially inserted by the input routine.) However, in this paper, for reasons of layout, it will be assumed that "newline" is not a character. ML/I treats text as a sequence of *atoms* rather than as a sequence of individual characters. An atom is defined as a single punctuation character or a sequence of letters and/or numbers surrounded by punctuation characters. Thus the piece of text which follows consists of the six atoms: comma, DOG, space, 23, plus, and C8.

```
,DOG 23+C8
```

Any sequence of atoms may be defined as the name of a construction or as the name of any of the other delimiters. In general every time the name of a macro is encountered during the scanning of text it is taken as the start of a macro call and a search is made for the remaining delimiters. The same applies to other constructions. Thus if DOG were a macro with closing delimiter plus, then the above text would contain a call of it. However, if DO were the macro name, the above text would not contain a call of it since DO is not an atom of the text.

The paragraphs which follow describe the other types of construction in addition to macros.

Inserts

Consider the replacement text of the MOVE macro which has already been used as an example. (This example and some subsequent ones will use PDP-7 Assembly Language. However, it is not assumed that the reader is familiar with the PDP-7 and each instruction will be explained.) The replacement text of the MOVE macro consists of the two following statements:

```
LAC argument 1   /Load accumulator with first argument.
DAC argument 2   /Deposit accumulator at second argument.
```

It is necessary to indicate to ML/I that it must insert the appropriate argument at the appropriate point. This is done by a construction called an *insert* which has a name and a closing delimiter. It will be assumed in the rest of this paper that the character ":" has been defined as an insert name with a period as its closing delimiter. Between the insert name and its delimiter a *designation* is written to indicate what to insert. In particular, "A" stands for argument. The replacement text of MOVE would be written:

```
LAC :A1.
DAC :A2.
```

Several other elements in addition to arguments may be inserted and these are described later.

Skips

It may be that the user does not wish certain occurrences of macro names in his text to be taken to mean the macro is to be called. This might apply, for instance, within comments or character string literals. In this case *skips* are used to inhibit macro replacement within the required context. Thus if the base language were PL/I a skip name "/*" with closing delimiter "*/" might be defined, since these characters are used to enclose PL/I comments. Each skip has an associated set of options which determine whether the skip is to be copied to the output text or deleted during macro expansion. It is possible to copy the delimiters and delete the intervening text or vice versa. In most applications the user will require a special skip name and closing delimiter called *literal brackets*. The options for literal brackets are set so that the brackets are deleted but the intervening text is copied literally to the output text. It will be assumed in the rest of this paper that the characters "⟨" and "⟩" have been defined as literal brackets. With this assumption, an occurrence of ⟨DOG⟩ in the source text would give rise to DOG in the output text irrespective of whether DOG was a macro name.

Warning Markers

In some applications it is inconvenient to have every occurrence of a macro name outside a skip taken as the beginning of a call. In these cases ML/I can be placed in *warning mode* by defining some atom as a *warning marker*. If ML/I is in warning mode all macro calls must be preceded by a warning marker and all macro names not preceded by a warning marker are treated literally. If ML/I

AD-A036 453 PURDUE UNIV LAFAYETTE IND PURDUE LAB FOR APPLIED IND--ETC F/G 9/2
SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIO--ETC(U)
JAN 77 N00014-76-C-0732

UNCLASSIFIED NL

4 OF 4
ADA036453

END
DATE
FILMED
3-77

is not in warning mode, macro names are essentially reserved words and if they are used for any other purpose they must be enclosed in skips. It will be assumed in examples in the rest of this paper that ML/I is not in warning mode.

Text Evaluation

The *environment* consists of all the constructions defined by the user, together with the system macros, which will be described later. The process of scanning a piece of text to expand macro calls and deal with skips and inserts is called *evaluating* the text. The text generated as a result of this evaluation is called the *value*. The value of a piece of text depends, of course, on the environment.

Constructions may be nested in any desired way. When ML/I encounters a macro call, it evaluates the replacement text in the same way as it evaluates the source text. The replacement text may itself contain macro calls and recursion is permitted. The arguments of macros are evaluated when they are inserted rather than when the call containing them is scanned. Thus they are "called by name" rather than "called by value." This fact is useful if ML/I is generating assembly code as it is possible to examine the context in which the code is to be placed and generate optimal code accordingly.

Macro Variables

ML/I contains some "macro-time" integer variables, i.e., variables operative during macro generation. There are facilities for the user to perform arithmetic on these variables and to insert their values into his text. There are two types of macro variables as follows.

Permanent variables. These are called P1, P2, etc. They are reserved at the start of a program and remain in existence throughout. They have global scope.

Temporary variables. These are called T1, T2, etc. Each time a user-defined macro is called, a number (defined by the user) of temporary variables is allocated. These are local to the current call. The first three temporary variables are initialized by ML/I in the following way:

T1 contains the number of arguments.

T2 contains the number of macro calls performed so far.

T3 contains the current depth of nesting of macro calls.

The initial value of T2 is a number unique to the current call and is useful for generating unique labels. If the replacement text of a macro contains a label it is imperative that a different name be generated for the label each time the macro is called. This can be achieved by writing the label as:

```
LAB:T2.
```

(Remember that the colon is assumed to be an insert name in this and all subsequent examples.) A later example illustrates the use of this feature.

Further Facilities for Inserts

It has been seen that arguments and macro variables may be inserted into text. This section describes the complete facilities offered by inserts.

The general form of the designation of an insert consists of a flag followed by a subscript expression. The subscript expression may consist simply of a positive integer, as in the previous examples, or it may be the name of a macro variable or an arithmetic expression involving macro variables and/or constants. Thus if T1 had its initial value then AT1 would reference the last argument and AT1-1 would reference the next to last argument. The following are the possible flags that can be used for inserts, together with a description of each:

A.	—argument.
D.	—delimiter.
WA, WD.	—argument or delimiter in its written form. The argument or delimiter is not evaluated but is inserted literally.
null.	—the numerical value of the subscript expression is inserted.
L.	—macro label. This type of insert "places" a macro-time label which may be the subject of a macro-time GO TO statement; it is a special type of insert in that it does not cause any output text to be generated, i.e., its value is null.

Operation Macros

ML/I contains a number of built-in macros called *operation macros*. Operation macros are used for such purposes as defining new constructions, performing macro-time arithmetic and changing the position of scan (using the macro-time GO TO statement). The names of operation macros all begin with the letters "MC" so that they may readily be differentiated from user-defined macros. An example of an operation macro is MCDEF, which is used for defining macros. This has form:

MCDEF *argument* 1 AS *argument* 2 ;

The first argument must be in the form of a *structure representation*, which specifies the delimiter structure of the macro being defined. In the simplest case, where all the delimiters are fixed, a structure representation is specified by writing the delimiters in the order in which they are to occur. The macro name, which is regarded as delimiter number zero, comes first. The second argument of MCDEF specifies the replacement text of the macro being defined. Thus if "TO" and semicolon were chosen to delimit the ends of the two arguments of MOVE, its definition would be written:

```
MCDEF MOVE TO ; AS
< LAC :A1.
  DAC :A2.
>;
```

and a call of form:

```
MOVE X TO TABLE + 6;
```

would expand into:

```
LAC X
DAC TABLE + 6
```

Note that the replacement text of MOVE has been enclosed in literal brackets in the above definition. This is because all arguments to operation macros are evaluated before being processed. Assume that the literal brackets had been omitted. Then in evaluating the second argument of MCDEF, ML/I would have tried to perform the inserts. This would have unfortunate results since the inserts should be performed when MOVE is called, not when it is defined. In its correct form, with the literal brackets present, the evaluation of the argument leads to the literal brackets being deleted and the enclosed text being copied literally. This text then becomes the replacement text of the MOVE macro. The reader is probably tempted to ask why operation macro arguments are not treated literally by the system in order to avoid the necessity of using literal brackets, but there are many examples where the dynamic element is vital.

Structure Representations for More Complicated Cases

The exact details of how to write structure representations in the general case are to be found in the User's Manual and this paper will confine itself to a basic outline. There are certain reserved words in structure representations. Among these are: WITH, OPT, OR, ALL and any atom consisting of the letter "N" followed by an integer. (The names of reserved words can be changed dynamically by the user, if he desires.) WITH is used to specify delimiters that consist of more than one atom. For example, if a delimiter consists of the atoms A1, A2, . . ., Ak, then this is specified by:

```
A1 WITH A2 WITH ... WITH Ak
```

The remaining reserved words are used to specify branches and nodes of the delimiter structure.

As has been seen, the purpose of a delimiter structure is to specify a successor or set of alternative successors for each delimiter. In a structure representation each specification of a delimiter is immediately followed by a specification of its successor. This successor may be defined in any of the following ways:

(a) If there is a single possible successor, as in the case of the delimiters of the MOVE example, this is specified simply by writing its name. (It is convenient to imagine a special symbol "null" occurs at the end of each structure representation. Any delimiter with "null" as its successor is a closing delimiter.)

(b) If there are alternative successors, this is specified by writing:

OPT *branch* 1 OR *branch* 2 OR *branch n* ALL

A branch may be a single delimiter specification or a structure in itself. An example of the use of this facility is the following macro to convert fully parenthesized algebraic notation to Polish Prefix notation.

```
MCDEF (OPT + OR - OR * OR / ALL) AS ⟨ :D1.:A1.:A2. ⟩;
```

Here the left parenthesis is the macro name. The replacement text consists of the first delimiter followed by the first argument followed by the second argument.

(c) If a delimiter is at the end of a branch, its successor is normally taken as the successor of the occurrence of ALL corresponding to the branch. Thus in the above Polish Prefix example each of the alternative arithmetic operators has a right parenthesis as its successor. However, a delimiter at the end of a branch can be given a different successor by writing the name of a *node* at the end of a branch. A node is designated by the letter "N" followed by a positive integer. The successor of the delimiter is then taken as the successor of the node, which is specified by writing the name of the node in front of some delimiter specification in the structure representation. The following example illustrates the use of nodes. Assume a macro called MIN allows any number of arguments separated by commas and terminated by a semicolon. Its structure representation would be written: MIN N1 OPT, N1 OR; ALL

This is interpreted thus. The successor of MIN is either a comma or a semicolon. Node N1 is to be associated with these alternatives since N1 has been written before the specification of the alternatives. If a comma is found as a delimiter, its successors are the successors of N1, namely either a comma or a semicolon. If a semicolon is found, its successor is the successor of ALL, namely "null."

Use of Macro-Time Statements

In simple macros such as MOVE, the replacement text is a fixed skeleton of code which is to replace each call. This is obviously not adequate for more sophisticated cases. Consider the IF macro used earlier as an example, which had alternative forms:

(a) IF *argument* 1 = *argument* 2 THEN *argument* 3 END
or (b) IF *argument* 1 = *argument* 2 THEN *argument* 3 ELSE *argument* 4 END

Clearly the replacement text of IF must be made to generate different code in the two cases. This can be done by using the macro-time GO TO statement, MCGO, in its form: MCGO *argument* 1 UNLESS *argument* 2 = *argument* 3;

The definition of IF is written as follows:

```
MCDEF IF = THEN OPT ELSE END OR END ALL AS
⟨        LAC :A1       /Load first argument.
         SAD :A2.      /Skip if it differs from the second
                         argument.
         SKP          /Equal case: skip one instruction
         JMP XX:T2.    /Unequal case: jump over inserted
                         code.
         :A3.          /Inserted code.
MCGO L1 UNLESS :D3. = ELSE; /Macro-time   jump   if
                              else clause is absent.
         JMP YY:T2.    /Jump over else clause.
```

```
XX:T2.,       :A4.              /Insert else clause.
YY:T2., MCGO L2;                /Macro-time jump to end of replace-
                                 ment text.
:L1.XX:T2., :L2. );             /Placing of macro-time labels.
```

Note that the second argument of MCGO, which is ":D3.", is evaluated before being compared with the character string "ELSE". This evaluation causes the text of the third delimiter to be inserted and to take part in the comparison. Note also the use of the initial value of T2 to generate unique labels. The following two successive calls of the IF macro:

```
IF A = B THEN JMS SUB
END
IF PIG = DOG THEN LAC C
                  DAC D
ELSE LAC Y
     DAC Z
END
```

would generate the code:

```
      LAC A        /First call.
      SAD B
      SKP
      JMP XX1
      JMS SUB
XX1,  LAC PIG      /Second call.
      SAD DOG
      SKP
      JMP XX2
      LAC C
      DAC D
      JMP YY2
XX2,  LAC Y
      DAC Z
YY2,
```

The two macro-time statements MCSET (the macro-time assignment statement) and MCGO can be used to form macro-time loops, which are useful in processing macros with a variable number of arguments. The following macro, which is useful in a number of applications, illustrates the macro-time loop. The macro, which has the composite name "INDEX (", successively compares its first argument with each succeeding argument until a match is found, and returns as its value the number of the matching argument.

```
MCDEF INDEX WITH (N1 OPT, N1 OR) ALL AS
(       MCSET T2 = 2;
:L2.    MCGO  L1 IF :AT2. = :A1.;
        MCSET T2 = T2 + 1;
        MCGO  L2;
:L1.    :T2. );     /Insert value of T2.
```

This macro can be used for switching. Assume it is desired to test the second delimiter of a macro and branch to macro-time label L2 if the delimiter is a plus sign, to L3 if it is a minus sign, and so on. Then the following statement achieves this:

```
MCGO L INDEX (:D2., +, -, *, /);
```

Scope of Definitions

New constructions may be defined at any time during text evaluation but, *ML/I being a one-pass system*, definitions only apply to text that comes after them. It is possible to redefine a construction and individual constructions may be deleted by redefining them to have a null effect. Constructions may be defined as local to a piece of replacement text or even local to the evaluation of an argument. It is possible to use a macro to set up the definition of another macro, and this is useful in dealing with declarative statements. Thus DECLARE could be a macro such that if, for example, the statement:

```
DECLARE X REAL
```

were encountered, then the DECLARE macro would set up a definition of X as a macro, which might, for instance, take the form:

```
MCDEFG X AS ( 100 MCSET P1 = 6 ; );
```

(MCDEFG is the same as MCDEF except that the definition is global rather than local to any containing macro.) This definition would cause each subsequent occurrence of X to be replaced by 100 and each call of X would, as a side-effect, set the permanent variable P1 to value six. The side-effect could be used to convey the information that X was of type "real".

Implementation

ML/I has been implemented on the PDP-7 and I.C.T. Atlas 2 computers. It occupies about three thousand words of storage. It is planned to write the logic of ML/I in a macro language so that, by suitably coding the basic macros and running the logical description through ML/I itself, it is possible to generate code for any machine. It is hoped that this technique will enable ML/I to be transferred to a new machine in about one man-month, even allowing for the multitude of unforeseen difficulties this kind of operation always involves.

RECEIVED JANUARY, 1967; REVISED MAY, 1967

REFERENCES

1. FAP: reference manual. Form C28-6235, IBM Programming Sys. Publ., Poughkeepsie, N.Y., Sept. 1962.
2. HALPERN, M. I. XPOP: a meta-language without metaphysics. Proc. AFIPS 1964 Fall Joint Comput. Conf., Vol. 26, pp. 57–68.
3. WILKES, M. V. An experiment with a self-compiling compiler for a simple list-processing language. *Annual Revue in Automatic Programming, Vol. 4.* Pergamon Press, Oxford, England, 1964.
4. WAITE, W. M. A language-independent macro processor. *Comm. ACM 10*, 7 (July, 1967), 433–440.
5. STRACHEY, C. A general purpose macrogenerator. *Comput. J. 8*, 3 (Oct. 1965), 225–241.
6. MOOERS, C. N. TRAC, a procedure describing language for the reactive typewriter. *Comm. ACM 9*, 3 (Mar. 1966), 215–219.
7. BROWN, P. J. ML/I user's manual. University Math. Lab., Cambridge, England, July 1966.

The Mobile Programming System: STAGE2

W. M. Waite
University of Colorado, Boulder, Colorado*

STAGE2 is the second level of a bootstrap sequence which is easily implemented on any computer. It is a flexible, powerful macro processor designed specifically as a tool for constructing machine-independent software. In this paper the features provided by STAGE2 are summarized, and the implementation techniques which have made it possible to have STAGE2 running on a new machine with less than one man-week of effort are discussed. The approach has been successful on over 15 machines of widely varying characteristics.

KEY WORDS AND PHRASES. *bootstrapping, macro processing, machine independence, programming languages, implementation techniques*
CR CATEGORIES: 4.12, 4.22

1. Introduction

In an earlier paper [1] R. J. Orgass and I presented a trivial macro processor called SIMCMP, which we proposed as a base for implementing a mobile programming system. The design criterion for SIMCMP was simplicity of implementation, and it was severely limited in the source code which it could accept. Its sole function was to translate a more complex processor. This paper describes STAGE2, the second level of the bootstrap.

STAGE2 is a flexible, powerful macro processor written in a language which can be translated by SIMCMP. It is designed specifically for the task of implementing machine-independent software, although it can also serve other purposes. Some features normally associated with a generalized macro processor have been omitted (for example, all macro definitions must be presented before any source code is read). These omissions are required because of restrictions imposed by SIMCMP on the number of program labels. They do not affect the primary function of STAGE2, and hence I feel that they do not represent design flaws.

To be effective as a tool for creating machine-independent software, STAGE2 itself must be highly mobile. For this reason, the support which the system requires from existing software on the target computer should be minimal. A generalized I/O package [2] provides the I/O interface for both SIMCMP and STAGE2. It assumes that the following operations on sequential files are provided by the operating system available on the target machine:

(1) read one line,
(2) detect end-of-file,
(3) write one line,
(4) write end-of-file,
(5) rewind.

These operations need only be available on those devices for which they are meaningful. The generalized I/O package carries flags which specify the legal operation on each device and will intercept any illegal requests.

In a minimal configuration, only three devices are required by STAGE2. Conceptually, they are a "card reader," "card punch," and "printer," but they may be implemented by any physical devices. It must be possible to perform operations (1) and (2) on the "card reader," and operation (3) on the "card punch" and "printer." Operation (4) can be used to clear the last buffer if the output is blocked. As described in Section 2, STAGE2 can use secondary storage (tape or disk) to advantage. A total of nine files, including the three just mentioned, would be the maximum. The six additional files can be used as secondary storage devices, additional inputs, or additional outputs. Generally all five operations would be possible on each, although this is not strictly necessary.

Implementation of STAGE2 on a new computer requires less than one man-week of effort. An existing version is unnecessary; the bootstrap process is carried out from a source deck, tape, or listing. The system has been tested in over 15 implementations, several completed by inexperienced personnel. The basic design philosophy of the system [3, 4] indicates a possible solution to the "software crisis" which is plaguing the computer industry today.

The remainder of this paper gives a brief overview of the facilities provided by STAGE2, and the programming techniques used in its construction. A comprehensive user's manual [5] is available for those interested in further details.

2. User's View

In many ways, STAGE2 is similar to LIMP [6]. It employs a scanning mechanism to recognize macro calls and isolate parameter strings, and the ability to perform different parameter conversions has been retained and extended. The code body does not include the "grouping" concept, nor LIMP's SNOBOL interpreter. Free use of I/O devices is permitted.

Each macro definition has two parts: a *template* and a *code body*. The template is used as a pattern for recognition of macro calls, and the code body directs the expansion of the macro. Both the template and the code body use special characters which are specified on the first line of the definitions (the *flag line*). Table I summarizes these characters.

---

* Department of Electrical Engineering

Template matching is carried out by a scanner which compares the input line with the templates. Characters in the template which are not parameter flags (Table I) must match the corresponding characters of the input line exactly. A parameter flag will match any substring of the input line which is balanced with respect to the parentheses specified by the flag line (a null string is considered to be balanced). By requiring that a parameter flag match a balanced substring of the input line, STAGE2 allows grouping by parentheses. This is a useful feature when structures such as arithmetic expressions or lists of operands are being analyzed.

TABLE I. CHARACTERS DEFINED BY THE FLAG LINE

Character Position	*Meaning*	*Character*
1	End of source language line	.
2	Parameter flag for template	'
3	End of target language line	$
4	Escape character for target language	#
5	Zero	0
6	Filler for fixed fields in the target language	space
7	Left parenthesis	(
8	Addition operator	+
9	Subtraction operator	-
10	Multiplication operator	*
11	Division operator	/
12	Right parenthesis	)

```
ALPHA = (BETA + GAMMA) * DELTA
```

(a) A typical input line

```
ALPHA = (') * DELTA
ALPHA = ' * '
ALPHA = '
' = '
' = (') * '
```

(b) Several templates which could match the input line given in Figure 1(a)

FIG. 1. Template matching

Figures 1(a) and 1(b) show an input line and several templates which would match it. The flag characters are those of Table I. Ambiguity in the template match is resolved by five rules (see Appendix). Together they have the effect of maximizing the number of literal characters matched. In Figure 1, for example, the rules force the input line to match the first template.

The actual parameters of a macro call are the substrings which match the parameter flags of the template. Within the code body, these actual parameters are referred to by number: the string matched to the leftmost flag is parameter 1, the one matched to the next flag is parameter 2, and so forth. A maximum of nine flags is allowed in a single template. The first template of Figure 1(b) has a single parameter flag. If the input line of Figure 1(a) were matched to this template, actual parameter 1 would be BETA + GAMMA. Actual parameters 2–9 would not be defined by the template match.

Associated with each template is a code body which specifies the text to be constructed by the macro. Each line of the code body is used to construct a line which is then either output (by means of a processor function, see (3) below) or else is resubmitted to the scanner for recognition. To construct the line, characters are copied from the code body line until the end-of-line (Table I) is recognized. The line is then terminated and passed to the scanner. If an escape character (Table I) is recognized during this process, it is not copied, and the next character of the line is examined to determine what action should be taken. There are three possibilities:

(1) If the next character is an escape or an end-of-line, then it is copied into the line being constructed.

(2) If the next character is a digit, then some transformation of the corresponding actual parameter is copied into the line being constructed. The particular transformation applied is determined by the character following the digit (see Table II). A transformation applied to parameter 0 is interpreted as a request for an arbitrary symbol which is unique within the current macro. Up to ten such symbols can be generated in any one macro. The second character following the escape specifies which symbol should be copied into the line being constructed.

(3) If the next character is F, then a special processor function is executed. The particular function is determined by the character following the F (see Table III).

STAGE2 provides three levels of storage: parameters, an associative memory, and up to six I/O files. The parameters can be used for working storage within a single macro. Information can be made accessible to all macros via the associative memory. Intermediate text can be written on any of the I/O files and later recovered.

Each macro call has storage space for nine parameters associated with it. The template match may establish initial values for some of these parameters. Parameter values may also be established by the code body, using transformation 6 (see Table II). The scope of the parameters is the code body of the macro in which they were established. They are not accessible to macros called by the code body. Their values are, however, preserved during such calls. On completion of a call, all parameters associated with that call disappear.

TABLE II. Parameter Conversion Digits

Conversion Digit	Action
0	Copy the parameter to the constructed line
1, 2	Copy a string from memory to the constructed line
3	Copy the break character to the constructed line
4	Copy the value of a parameter, treated as an arithmetic expression, into the constructed line
5	Copy a parameter length to the constructed line
6	Reset the value of a parameter
7	Initiate a context-controlled iteration
8	Copy an integer equivalent to a single character into the constructed line

TABLE III. Processor Function Digits

Function Digit	Action
0	Terminate processing
1	Output a line without rescanning
2	Change I/O channels and copy text
3	Store information into memory
4	Set skip counter unconditionally
5	Set skip counter conditionally on string equality
6	Set skip counter conditionally on the relative values of two expressions
7	Initiate a count-controlled iteration
8	Advance an iteration
9	Escape from the current macro
E	Generate error traceback

The associative memory is addressed by character strings, and its contents are also character strings. No restriction is placed on the format of either addresses or contents. Information is entered into memory by a processor function, and accessed by parameter transformations. Because it is global to all macro calls, the memory can be used to pass information between macros. Typical uses of the memory are to hold a symbol table, a pushdown stack for translating expressions, and information about the current contents of various registers for use in "peephole" optimization [7].

A maximum of nine I/O files can be made accessible to STAGE2. At least three (the "reader," "punch," and "printer") will always be available. By suitable processor functions, the user can direct output generated by STAGE2 to any file on which writing is allowed. The input may be switched to any file on which reading is allowed. Any file may be rewound if rewinding is allowed by its implementation.

The main use of I/O files is to permit multipass operation. For example, complex code optimization, such as register allocation [8, 9], can be performed using the available I/O files for temporary storage of intermediate text. Information about register contents can be accumulated in the memory between labels, while intermediate text is written on one or more files. At the next label, this information is examined and a register assignment made. The intermediate text is then read back and the final code is generated.

STAGE2 can evaluate integer arithmetic expressions involving addition, subtraction, multiplication, and division. Parentheses may be used to any depth (subject to storage availability). The operands may be either explicit integers or symbols. A symbol is looked up in the associative memory, where it must have an integer value.

Conditional operations are available for testing the equality of two strings and the relative magnitude of two expressions. If the condition is satisfied, an expression is evaluated to determine how many lines should be skipped. A skip is not restricted to the macro in which the conditional operation appeared.

It is possible to loop within a code body. The loop is controlled either by a counter or by a string and a set of delimiters. Using the latter, known as *context controlled* iteration, it is possible to analyze complex parameters such as lists of operands and arithmetic expressions. The context controlled iteration is, in effect, a token breakout routine similar to that found in a compiler. No particular delimiters are assumed; they are provided when the iteration is initiated.

Tables II and III summarize the parameter transformations and special processor functions available to the user of STAGE2. The brief discussion presented in this section is intended to emphasize the main features of the processor and sketch the broad outlines of its operation. For full details, the interested reader is referred to the user's manual [5].

3. Implementation

The programming techniques used in writing STAGE2 were chosen to make it easy to implement on virtually any machine. It is coded in the assembly language of a special purpose computer, called Flub, which was designed to handle the data structures relevant to macro processing: trees, strings, and integers. Flub does not exist; it is an abstract machine [3] conceived solely for the purpose of coding STAGE2. (A complete description of Flub, giving details of the rationale behind the design, can be found in [10].) To implement STAGE2 on a real computer, the Flub operations are realized as macros which can be expanded by SIMCMP [1]. Flub has 28 machine operations and 2 pseudo operations, so that a total of 30 macros must be written.

The general organization of the Flub computer is shown in Figure 2. Each register is made up of three fields, as described in Table IV. The instruction set is given in Table V. Note that many instructions which are commonplace on computers today are not included. Shift instructions and Boolean operations, for example, are conspicuous by their absence, the reason being that STAGE2 does not require such operations. It is important to keep the purpose

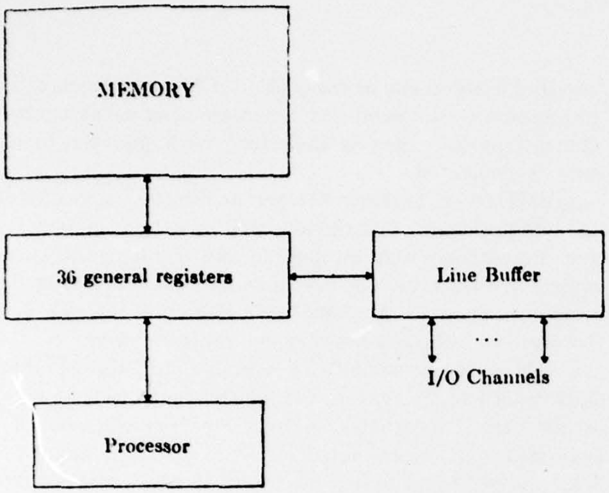


Fig. 2. Organization of the Flub computer

TABLE IV. Breakdown of the Flub Word

Field	*Minimum Range of Values*	*Purpose*
FLG	0, 3	Indicator bits
VAL	0, max(C, L) + 1	To hold a single character or the length of a string. C is the largest integer which represents a character, L is the maximum string length
PTR	−A, A	To address Flub memory, and act as a general integer accumulator. A is the largest address of the target machine's memory

TABLE V. Flub Machine Instructions

1. Data Transfer Operations

(a) Register to Register	(b) Register to Memory
FLG ' = '.	GET ' = '.
VAL ' = PTR '.	STO ' = '.
PTR ' = VAL '.	

2. Integer Arithmetic Operations

(a) VAL field arithmetic	(b) PTR Field arithmetic
VAL ' = ' + '.	PTR ' = ' + '.
VAL ' = ' − '.	PTR ' = ' − '.
	PTR ' = ' * '.
	PTR ' = ' / '.

3. Input/Output Operations

(a) Character I/O	(b) Line I/O
VAL ' = CHAR.	READ NEXT '.
CHAR = VAL '.	WRITE NEXT '.
	REWIND '.
	MESSAGE '' TO '.

4. Control Operations

(a) Unconditional	(b) Conditional
STOP.	TO '' IF FLG ' = '.
TO ''.	TO '' IF FLG ' NE '.
TO '' BY '.	TO '' IF VAL ' = '.
RETURN BY '.	TO '' IF VAL ' NE '.
	TO '' IF PTR ' = '.
	TO '' IF PTR ' NE '.
	TO '' IF PTR ' GE '.

of this abstract machine firmly in mind when criticizing omissions in the instruction set. Flub makes no pretense about being a computer for general problems; it is specifically tailored to run STAGE2.

All Flub operands are register names, except target locations for jump instructions and the first operand of MESSAGE '' TO '. Each of these consists of two digits. (GET and STO each specify a register which contains the desired memory address.) Each apostrophe in Table V thus represents a single letter or digit. This conforms to the restriction on parameters which is imposed by SIMCMP, and in fact the lines listed in Table V are exactly the templates for the SIMCMP macros used to translate STAGE2.

An advantage of performing all Flub operations on registers is that it is possible to make a realization of STAGE2 more efficient by using different storage techniques for registers and memory. On one computer, the Titan at Cambridge, it was possible to assign flip-flop registers to all of the Flub registers. Even if the Flub registers are stored in core, their fields can be left unpacked for rapid access. The fields of the memory words would be packed to conserve space.

All Flub I/O is handled through a generalized I/O interface [2] which provides access to a number of *channels* via a *line buffer*. Information is transmitted one character at a time between the line buffer and the general registers. Full lines are transmitted between the line buffer and the channels. This view of I/O is quite similar to that proposed for Algol 60 [11, 12] and being implemented for Algol 68 [13].

The code bodies of the macros whose templates are shown in Table V, the I/O interface, and SIMCMP are machine-dependent. They must be hand-coded for each implementation, a task which requires about 8 man-hours. We have a set of macros whose code bodies are written in ASA Fortran, and an ASA Fortran version of the I/O package. SIMCMP is also described in ASA Fortran [1]. Thus, if Fortran is available on the target computer, initial implementation of STAGE2 is trivial.

The initial implementation of STAGE2 is via SIMCMP. Because of the inherent weakness of SIMCMP, this implementation will generally not be extremely efficient. To improve the efficiency, one would rewrite the 30 macros to take advantage of the powerful facilities which STAGE2 provides for optimization. Then, using these macros and the initial implementation of STAGE2, it is possible to retranslate STAGE2.

Still further improvements in efficiency can be obtained by hand-coding certain critical routines. When STAGE2 is translated by either SIMCMP or STAGE2, the result is an assembly code program. It is possible to make measurements on the performance of this program to determine where it is spending most of its time. The critical routines can then be recoded, taking advantage of all available programming tricks. The effort involved in this recoding is miniscule compared with that involved in hand-coding the entire processor.

4. Conclusions

The purpose of STAGE2 is to provide a tool for constructing machine-independent software. STAGE2 itself is machine-independent, and was designed in such a way that it could be moved to a new machine with virtually no effort and without relying on existing software. I believe that these objectives have been met. As noted in the Introduction, the mobility of STAGE2 has been proved in over 15 implementations. In fact, in some cases we have found that reimplementation from scratch was simpler than trying to bring up an existing version for the same machine from a different installation!

One of the major questions which arises with a machine-independent implementation such as that described in this paper is how much it costs. What penalty is assessed in terms of reduced efficiency in both time and space? Unfortunately, I cannot answer that question conclusively. To do so would require an implementation of STAGE2 "by hand" for comparison purposes. As an indication, however, consider the statistics of Table VI.

TABLE VI. IMPLEMENTATION FOR CDC 6000 SERIES

	I. Environment	
Generalized I/O package	495	words
I/O buffers	3019	
System routine	1622	
Data space	12000	
Total:	17136	words
	II. STAGE2	
FLUB source code	977	statements
Unoptimized version	2373	words
Optimized version	1667	words
	III. Running time to translate STAGE2 (6400)	
Unoptimized version	42.254	seconds CP
	7.781	seconds PP
Optimized version	36.360	seconds CP
	7.233	seconds PP

These numbers were obtained from STAGE2, version 2, as implemented on the CDC 6400 at the University of Colorado. At the time the runs were made, we were operating under the standard release of SCOPE 3.1.6. The generalized I/O package was tailored to SCOPE and the 6400, and the system routines were unmodified. I/O buffers were provided for nine channels, using the minimum number of words to allow double buffering with the device type assigned to the channel (129 words for disk files, 1025 words for tape files). Note that with this environment, STAGE2 itself represents less than 13 percent of the total core requirement.

In the unoptimized version, each FLUB instruction was translated to COMPASS in the most straightforward way. No attempt was made to take advantage of special operands or to minimize instruction length by retaining frequently used constants in registers. For example, the FLUB statement

$$\text{PTR} \quad A = 0 + 0$$

was translated into a fetch of the PTR field of register 0, fetch and add of the same field, and finally a store into the PTR field of A. The optimizing macros took advantage of such operands. In the case noted above, a zero was stored in PTR A (the FLUB register 0 contains 0 in all of its fields). They provide for certain frequently used constants (such as 0 and 1) to be held in registers at all times, and recognize other machine-dependent situations where 15-bit instructions can be used instead of 30-bit instructions. No global optimization is performed; only information derived from a single macro call is examined.

Note that although the optimization provided a significant 30 percent reduction in the size of STAGE2, the *total* length of the optimized version was still 97 percent of that of the unoptimized version. Optimization also provided somewhat faster code—the optimized version required only about 86 percent of the time used by the unoptimized version. Both programs were given the same task: translating STAGE2 to COMPASS using the optimizing macros. Since the FLUB source code for STAGE2 is 977 statements long, STAGE2 translates at roughly 1600 cards per minute with these macros.

To date, STAGE2 has been used to translate a comprehensive text processor known as MITEM [14], and a small, interactive editor called MICE. Both of these programs have been implemented on several computers with less than one man-week of effort. We are currently writing a page layout and printing program similar to Text/360 [15] and FORMAT [16], and an interactive BASIC [17] system.

I believe that the techniques used in implementing STAGE2, and STAGE2 itself, can help to alleviate the software crisis in which we find ourselves today. By now, most people have learned the value of higher level languages for machine independence. The difficult areas are those in which no high level language is quite suited to the problem. There are two possible tacks to take: (1) twist the problem to fit some existing language; (2) design another language. The major objection to (2) has been the difficulty of implementing a new language, and the realization that the processor will be useless on any other machine. STAGE2 removes the latter constraint. A processor is still difficult to construct, but once done it can be used almost anywhere.

Appendix. Template Matching

The templates are organized into a *tree*, with each pattern element (single character or parameter flag) corresponding to a *branch*. An example of a template tree is shown in Figures 3(a) and 3(b). Uppercase letters and special characters denote themselves. (A space appears as no character at all.) A lowercase c denotes the end-of-line marker, and a lowercase p the parameter flag. The reason for using c and p is that these branches are not normal characters. As the templates are being read in, each end-of-line flag (or carriage return, if the end-of-line flag is absent) is replaced by a special marker. Each parameter flag is likewise replaced by a marker of another type. If an input line has no end-of-line flag, then the carriage return is replaced by the c marker. Thus there is no internal distinction between an end-of-line flag and a carriage return.

Dashed lines in Figure 3(b) represent *nodes* of the tree. Each branch (character) has a direction associated with it--left to right in Figure 3(b). A given node may have any number of branches directed away from it, but not more than one directed toward it. One node, known as the *root*, has no branches directed toward it. Several other nodes have no branches directed away from them. These are the *leaves* of the tree, and are numbered to correspond with the templates. When an input line matches the tree from the root to some leaf, it is recognized as an instance of the template corresponding to that leaf.

In Figure 3(b), the branches leaving a node are arranged vertically. Thus the root has two branches leaving it, an S and a parameter flag. The node toward which S is directed has only one branch leaving it—the character A. We say that one node can be *reached* from another by a branch if the branch is directed from the first node to the second. Hence the node between S and A can be reached from the root by S.

(1) SAM = A.
(2) SAM = '.
(3) ' = '.
(4) ' = A.
(5) ' = ' = '.
(6) SAM = JOE.

FIG. 3(a). A set of templates

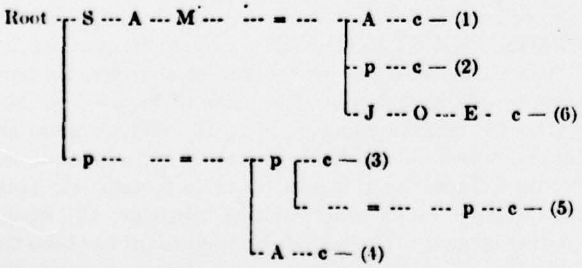

Root
S --- A --- M --- . --- = --- A --- c — (1)
p --- c — (2)
J --- O --- E - c — (6)
p --- --- = --- p --- c — (3)
--- = --- --- p --- c — (5)
A --- c — (4)

FIG. 3(b). The tree built from the templates of Figure 3(a)

When an input line is read, the end-of-line flag is replaced by a termination element. If there is no end-of-line flag, then the carriage return is replaced. Parameter flag characters are not recognized when input lines are being read; they are only significant in definitions. The input line is matched against the template tree according to the following rules.

Rule 1. Rule 2 is applied to the root of the tree and the first character of the input line.

Rule 2. If one of the branches leaving the current node of the tree is identical to the current input character, then that branch matches the current input character. Otherwise, Rule 3 is applied to the current node and input character. If the branch matched is an end-of-line marker, the input line is recognized. Otherwise Rule 2 is applied to the node reached from the current node by that branch and the next character of the input line.

Rule 3. If one of the branches leaving the current node of the tree is a parameter flag, then Rule 2 is applied to the node reached from the current node by that branch, and the current input character. The branch matches a null string. Otherwise, Rule 4 is applied to the current node.

Rule 4. If the current node is the root, the match fails. Otherwise, if the branch entering the current node is not a parameter flag, then Rule 3 is applied to the previous node and the input character which matched the entering branch. Otherwise Rule 5 is applied to the current node and current input character.

Rule 5. The substring matched by the branch entering the current node is lengthened by appending the shortest balanced substring of the input line which begins at the current input character. Rule 2 is then applied to the current node and the input character following the new substring. If no such substring can be found, Rule 4 is applied to the previous node and the first input character matched by the parameter flag. If the parameter flag matched the null string, the current input character is used.

If the match fails, the input line is written out. Otherwise the substring which matched the leftmost parameter flag is made parameter 1, the substring which matched the next parameter flag becomes parameter 2, and so forth. (A maximum of nine parameter flags is allowed in one template.) When the parameters have been set, interpretation of the code body begins.

As an example of the matching process, consider the set of templates of Figure 3(a) and the corresponding tree of Figure 3(b). The input line SAM = A will obviously match template 1. It could also match template 2, with A the value of parameter 1, template 3 with SAM and A as parameters, or template 4 with SAM matching the parameter flag. Let us see how the rules operate to determine which of these possibilities will actually occur. Rule

1 gets the whole process started by calling for application of Rule 2 to the root and the first character of the input string. The first character of the input string is S, and there is an S branch directed away from the root of the tree. Thus Rule 2 matches S to this branch. The branch matched is not an end-of-line marker. Hence Rule 2 is applied to the node reached by that branch, and the next character of the input string. In our case the next input character happens to be A, and there is an A branch directed away from the node reached by S from the root. Since this branch is not an end-of-line marker, Rule 2 is applied again. The process continues, with Rule 2 finding a match at each node. Finally, it matches the terminating element of the line to the c branch, and recognizes an instance of template 1.

At first glance, one might assume that the input line B = C = A would match template 4 with B = C as the value of the actual parameter. This is not true, however, as an analysis of the scan will show. Rule 1 starts us off as above, but this time Rule 2 does not produce a match and we must apply Rule 3. There is a parameter marker. Rule 2 is applied to the node reached by this branch and the current input character (B). Rule 2 fails to produce a match here also, so Rule 3 is tried. Unfortunately, there is no parameter marker leaving this node, so that Rule 3 sends us on to Rule 4. The current node is not the root but the branch entering the current node is a parameter flag. Rule 5 is therefore used to extend the null string which originally matched this flag. B is a balanced substring, so that we now apply Rule 2 to the current node and the space character which follows B. This time Rule 2 produces a match, and we can proceed. The = and the second space are also matched by Rule 2, but then Rule 2 cannot find a match for C. Thus Rule 3 is used to advance along the parameter branch, matching the parameter to the null string. We will now have the same situation which occurred at the first parameter marker—Rules 2 and 3 will be unable to cope with the following node, and eventually Rule 5 will be called upon to extend the substring matched by the parameter flag. Rule 2 will then find a match for the space following the C, and the whole process will repeat itself. The A will finally be matched to the last parameter flag, and the matching of the terminator will complete the process. The line will thus be recognized as an instance of template 5 rather than template 4.

Let us now consider the input line (B = C) = A. This is the same as that of the previous example, except that parentheses have been added to group the characters B = C together. The scan of this line will begin in exactly the same way as the scan of B = C = A. This time, however, when Rule 5 is applied to extend the substring of the input which begins at the first input character is (B = C). We now apply Rule 2 to the node following the parameter marker and a space character from the input string. There is a match, so Rule 2 is applied to the next node and the = character. This matches also, as does the space. We must now apply Rule 2 to the node reached by the space and the input character A. Notice here that there *is* a branch leaving the node which matches A. Thus Rule 2 is applied again to the node reached by that branch and the terminating element of the input line. We have a match here, too, and Rule 2 therefore states that the match is successful. Because of the fact that a parameter flag must match a balanced substring of the input line, (B = C) = A is recognized as an instance of template 4, and (B = C) will be the value of the first actual parameter. If the line were (B = C) = D = A, the reasoning of the previous paragraph shows that it would match template 5.

The template-matching scheme described in this Appendix permits flexibility in the format of a macro call. It accommodates varying language styles but does require that only one statement be written per line. Spaces are significant. The fact that each parameter flag matches a balanced substring of the input line allows for grouping of text by parentheses.

Received December, 1969

REFERENCES

1. Orgass, R. J., and Waite, W. M. A base for a mobile programming system. *Comm. ACM 12*, 9 (Sept. 1969), 507–510.
2. Poole, P. C., and Waite, W. M. I/O for a mobile programming system. Tech. Rep. 69-1, Graduate School Computing Center, U. of Colorado, Boulder, Colo.
3. Poole, P. C., and Waite, W. M. Machine independent software. Proc. ACM Second Symp. on Operating Systems Principles. Dep. of Elec. Eng., Princeton U., Princeton, N. J., Oct. 1969.
4. Waite, W. M. Building a mobile programming system. *Comput. J. 13*, (Feb. 1970), 25–31.
5. Waite, W. M. The STAGE2 macro processor. Tech. Rep. 69-3, Graduate School Computing Center, U. of Colorado, Boulder, Colo.
6. Waite, W. M. A language independent macro processor. *Comm. ACM 10*, 7 (July 1967), 433–440.
7. McKeeman, W. M. Peephole optimization. *Comm. ACM 8*, 7 (July 1965), 443–444.
8. Horwitz, L. P., Karp, R. M., Miller, R. E., and Winograd, S. Index register allocation. *J. ACM 13*, 1 (Jan. 1966), 43–61.
9. Lowry, E. S., and Medlock, C. W. Object code optimization. *Comm. ACM 12*, 1 (Jan. 1969), 13–22.
10. Waite, W. M. Building a mobile programming system. Tech. Rep. 69-2, Graduate School Computing Center, U. of Colorado, Boulder, Colo.
11. IFIP/WG 2.1. Report on input-output procedures for ALGOL 60. *Comm. ACM 7*, 10 (Oct. 1964), 628–630.
12. DeVogelaere, R. A set of basic input-output procedures. *Comm. ACM 11*, 8 (Aug. 1968), 567–573.
13. Mailloux, B. J. Private communication.
14. Poole, P. C., and Waite, W. M. A machine independent program for the manipulation of text (User manual). Tech. Rep. 69-4, Graduate School Computing Center, U. of Colorado, Boulder, Colo.
15. Reed, S. L. TEXT360. 360D 29.5.002, IBM Program Library, Aug. 1967.
16. Berns, G. M. Description of FORMAT, a text processing program. *Comm. ACM 12*, 3 (Mar. 1969), 141–146.
17. Dartmouth College Computer Center. *BASIC* (3rd ed.). Dartmouth CCC, Hanover, N. H., 1966.